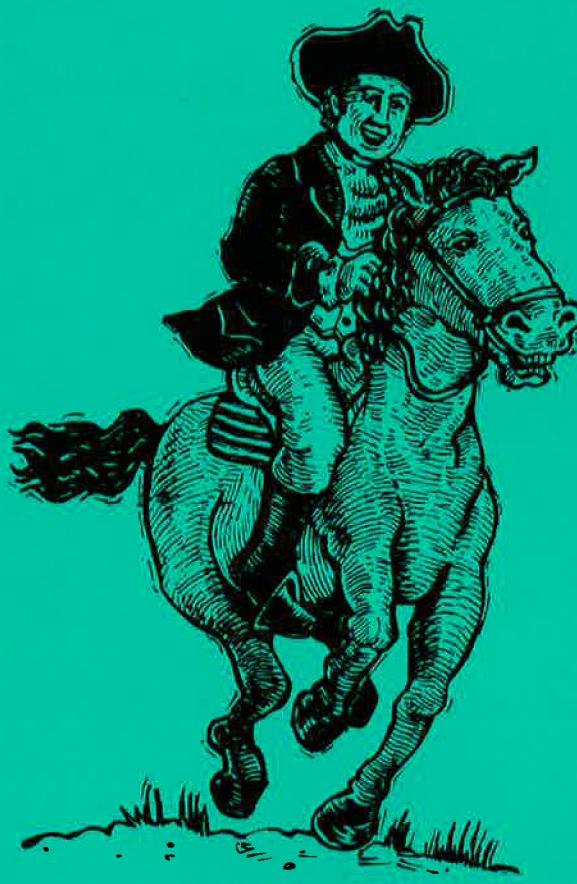


Conference Proceedings

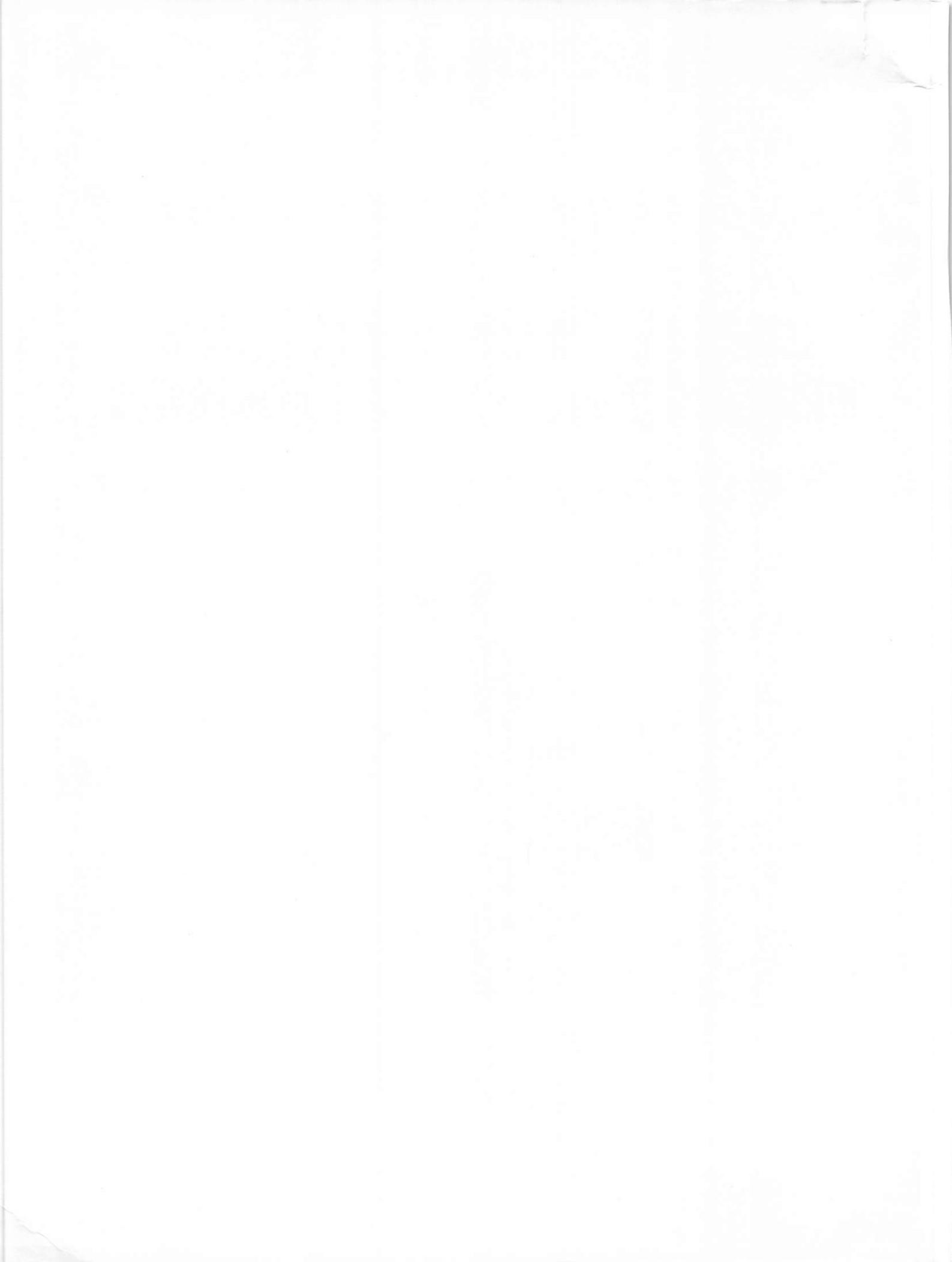
USENIX SUMMER 1994 TECHNICAL CONFERENCE

Boston, Massachusetts • June 6–10, 1994



USENIX®

THE UNIX® AND ADVANCED COMPUTING SYSTEMS PROFESSIONAL AND TECHNICAL ASSOCIATION



USENIX Association

**Proceedings of the
Summer 1994 USENIX Conference**

**June 6 - 10, 1994
Boston, Massachusetts, USA**

For additional copies of these proceedings write:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA

The price is \$25 for members and \$33 for nonmembers.
Outside the U.S.A. and Canada, please add
\$18 per copy for postage (via air printed matter).

Past USENIX Technical Conferences

1994 Winter San Francisco	1988 Winter Dallas
1993 Summer Cincinnati	1987 Summer Phoenix
1993 Winter San Diego	1987 Winter Washington, DC
1992 Summer San Antonio	1986 Summer Atlanta
1992 Winter San Francisco	1986 Winter Denver
1991 Summer Nashville	1985 Summer Portland
1991 Winter Dallas	1985 Winter Dallas
1990 Summer Anaheim	1984 Summer Salt Lake City
1990 Winter Washington, DC	1984 Winter Washington, DC
1989 Summer Baltimore	1983 Summer Toronto
1989 Winter San Diego	1983 Winter San Diego
1988 Summer San Francisco	

1994 © Copyright by The USENIX Association
All Rights Reserved.

ISBN 1-880446-62-6

This volume is published as a collective work.
Rights to individual papers remain with the author or the author's employer.

USENIX acknowledges all trademarks herein.

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste. 

TABLE OF CONTENTS

USENIX Summer 1994 Technical Conference
June 6 - 10, 1994
Boston, Massachusetts, USA

OFF THE BEATEN TRACK

Wednesday 11:00 - 12:30

Chair: Glen Dudek, VGI

A New Object-oriented Programming Language: <i>sh</i>	1
<i>Jeffrey S. Haemer, Canary Software, Inc.</i>	

The Old Man and the C.....	15
<i>Evan Adams, Sun Microsystems</i>	

CLOSING THE SYSTEM

Wednesday 2:00 - 3:30

Chair: Greg Rose, RoseSecure

Key Management in an Encrypting File System	27
<i>Matt Blaze, AT&T Bell Laboratories</i>	

A Toolkit and Methods for Internet Firewalls.....	37
<i>Marcus K. Ranum, Frederick M. Avolio, Trusted Information Systems, Inc.</i>	

SNP: An Interface for Secure Network Programming.....	45
<i>Thomas Y.C. Woo, Raghuram Bindignavle, Shaowen Su, and Simon S. Lam, University of Texas, Austin</i>	

UNDER THE COVERS

Wednesday 4:00 - 5:30

Chair: Adam Moskowitz, Menlo Consulting

An Efficient Kernel-Based Implementation of POSIX Threads	59
<i>Robert A. Alfieri, Unix Kernel Development, Data General Corporation</i>	

Using OS Locking Services to Implement a DBMS: An Experience Report.....	73
<i>Andrea H. Skarra, AT&T Bell Laboratories</i>	

The Slab Allocator: An Object-Caching Kernel Memory Allocator	87
<i>Jeff Bonwick, Sun Microsystems</i>	

FILE SYSTEMS

Thursday 9:00 - 10:30

Chair: Kent Peacock

A Better Update Policy	99
<i>Jeffrey C. Mogul, Digital Equipment Corporation, Western Research Laboratory</i>	

The Desktop File System	113
<i>Morgan Clark, Stephen Rago, Programmed Logic Corporation</i>	

Sawmill: A High Bandwidth Logging File System	125
<i>Ken Shirriff, John Ousterhout, University of California, Berkeley</i>	

THE NETWORK STRIKES BACK

Thursday 11:00 - 12:30

Chair: Win Treese, Digital Equipment Corporation

NFS Version 3: Design and Implementation	137
<i>Brian Pawlowski, Network Appliance, Chet Juszczak, Digital Equipment Corporation, Peter Staubach, SunSoft Inc., Carl Smith, SunSoft Inc., Diane Lebel, Digital Equipment Corporation, Dave Hitz, Network Appliance</i>	
Clue Tables: A Distributed, Dynamic-Binding Naming Mechanism	153
<i>Cheng-Zen Yang, Chih-Chung Chen, Yen-Jen Oyang, National Taiwan University</i>	
Optimistic Lookup of Whole NFS Paths in a Single Operation	161
<i>Dan Duchamp, Columbia University</i>	

REVENGE OF THE FILE SYSTEMS

Thursday 2:00 - 3:30

Chair: Mary Baker, Stanford University

Application-controlled File Caching Policies	171
<i>Pei Cao, Edward W. Felten, Kai Li, Princeton University</i>	
Resolving File Conflicts in the Ficus File System	183
<i>Peter L. Reiher, John Heidemann, David Ratner, Gregory Skinner, Gerald J. Popek, University of California, Los Angeles</i>	
Reducing File System Latency using a Predictive Approach	197
<i>Jim Griffioen, Randy Appleton, University of Kentucky</i>	

POTPOURRI

Friday 9:00 - 10:30

Chair: Michael Jones, Microsoft Research

Operating System Support for Distributed Multimedia	209
<i>Sape J. Mullender, Ian M. Leslie, Derek McAuley, University of Twente</i>	
Splicing UNIX into a Genome Mapping Laboratory	221
<i>Lincoln Stein, Andre Marquis, Robert Dredge, Mary Pat Reeve, Mark Daly, Steve Rozen, Nathan Goodman, Whitehead Institute for Biomedical Research</i>	
A Text Retrieval Package for the Unix Operating System	231
<i>Liam R. E. Quin, SoftQuad Inc.</i>	

OPENING CLOSED SYSTEMS

Friday 11:00 - 12:30

Chair: Ozan Yigit, York University

Probing TCP Implementations	245
<i>Douglas E. Comer, John C. Lin, Purdue University</i>	
Experiences with a Survey Tool for Discovering Network Time Protocol Servers.....	257
<i>James D. Guyton, Michael F. Schwartz, University of Colorado, Boulder</i>	
Profiling and Tracing Dynamic Library Usage Via Interposition	267
<i>Timothy W. Curry, Sun Microsystems</i>	

GETTING AROUND

Friday 2:00 - 3:30

Chair: Peter Honeyman, CITI University of Michigan

Large Granularity Cache Coherence for Intermittent Connectivity.....	279
<i>Lily B. Mummert, M. Satyanarayanan, Carnegie Mellon University</i>	
An Analysis of Trace Data for Predictive File Caching in Mobile Computing	291
<i>Geoffrey H. Kuenning, Gerald J. Popek, Peter L. Reiher, University of California, Los Angeles</i>	
Secure Short-Cut Routing for Mobile IP	305
<i>Trevor Blackwell, Kee Chan, Koling Chang, Thomas Charuhas, James Gwertzman, Brad Karp, H. T. Kung, David Li, Dong Lin, Robert Morris, Rob Polansky, Diane Tang, Cliff Young, John Zao, Harvard University</i>	

Reviewers

Lee Damon, *Qualcomm*
Chris Fraser, *AT&T Bell Laboratories*
Dan Geer, *OpenVision Technologies*
J. S. Goldberg, *The Harper Group*
Richard Golding, *Hewlett Packard Laboratories*
Robert Gray, *U. S. West*
Frank Greco, *Mercury Technologies, Inc.*
Brian Kernighan, *AT&T Bell Laboratories*
Christopher J. Lefelhocz, *MIT Laboratory for Computer Science*
John LoVerso, *Open Software Foundation*
Susan LoVerso, *Thinking Machines Corporation*
Larry McVoy, *Sun Microsystems*
George Neville-Neil, *Xaos Tools*
Joann J. Ordille, *AT&T Bell Laboratories*
Andrew Payne, *Digital Equipment Corporation*
Dave Presotto, *AT&T Bell Laboratories*
Rich Salz, *Open Software Foundation*
Stuart Shieber, *Harvard University*
Christopher Small, *Harvard University*
Keith A. Smith, *Harvard University*
Michael D. Smith, *Harvard University*
Karen R. Sollins, *MIT Laboratory for Computer Science*
Carl Staelin, *Hewlett Packard Laboratories*
Larry Stewart, *Digital Equipment Corporation*
Phil Winterbottom, *AT&T Bell Laboratories*
Cliff Young, *Harvard University*

Acknowledgements

Program Chairs

Keith Bostic, *U. C. Berkeley*
Margo Seltzer, *Harvard University*

Program Committee

Mary Baker, *Stanford University*
John Bongiovanni, *Stratus Computer*
Glen Dudek, *VGI*
Peter Honeyman, *CITI, University of Michigan*
Michael B. Jones, *Microsoft Research*
Adam S. Moskowitz, *Menlo Consulting*
Kent Peacock, *Consultant*
Rob Pike, *AT&T Bell Laboratories*
Greg Rose, *Australian Computing and Communications Institute*
Win Treese, *Digital Equipment Corp.*
Ozan S. Yigit, *York University; SIS Project*

Program Committee Scribe

Cliff Young, *Harvard University*

Proceedings Production

Carolyn Carr, *USENIX Association*
Malloy Lithographing, Inc.

Invited Talks

Robert Gray, *US West*
Brent Welch, *Xerox PARC*

Tutorial Program

Daniel Klein, *USENIX Association*

25th UNIX Anniversary Events

Peter Salus, *Consultant*

Works in Progress Coordinator

Peg Shafer, *BBN*

USENIX Board Liaison

Evi Nemeth, *University of Colorado, Boulder*

Vendor Display

Peter Mui, *USENIX Association*

Administration

The USENIX Association Staff

Publicity

Cynthia Deno, *USENIX Association*

Terminal Room

Gretchen Phillips, *University of Buffalo*

USENIX Executive Director

Ellie Young, *USENIX Association*

The GURU is In Coordinator

Ed Gould, *Digital Equipment Corporation*

USENIX Meeting Planner

Judy DesHarnais, *USENIX Association*

Preface

Welcome to the USENIX Summer 1994 Conference!

Thanks to the bush-beating efforts of the Program Committee, we received more than a hundred paper submissions. We hope that you find the results both interesting and educational.

This conference has certainly been “interesting and educational” for us! It would not have been possible without a Program Committee that went far beyond the call of duty, and the help of the many outside reviewers. Evi Nemeth was a great help as Board Liaison. Brent Welch and Bob Gray designed the Invited Talks track, and Bob served on the Program Committee as well. Peter Salus orchestrated the historical events, Dan Klein put the tutorial track together, Peg Shafer coordinated the Works-in-Progress session and Ed Gould arranged the “Guru is in” sessions. And, of course, the staff at Usenix (that have to put up with the whims of new program chairs every six months) continue to do a fabulous job. Carolyn Carr put the proceedings together, Cynthia Deno organized and produced the Call for Papers and Programs, Judy DesHarnais handled all of the conference logistics, Toni Veglia managed the rest of the details, and Ellie Young ran the show. If you see these people at the conference, please offer them your thanks; they make the Usenix conferences happen.

We wish you an exciting 25th Anniversary and a great conference.

Margo Seltzer
Keith Bostic
Program Co-Chairs

Author Index

Evan Adams	15	David Li	305
Robert A. Alfieri	59	John C. Lin	245
Randy Appleton	197	Dong Lin	305
Frederick M. Avolio	37	Andre Marquis	221
Raghuram Bindignavle	45	Derek McAuley	209
Trevor Blackwell	305	Jeff C Mogul	99
Matt Blaze	27	Robert Morris	305
Jeff Bonwick	87	Sape J. Mullender	209
Pei Cao	171	Lily B. Mummert	279
Kee Chan	305	John Ousterhout	125
Koling Chang	305	Yen-Jen Oyang	153
Thomas Charuhas	305	Brian Pawlowski	137
Chih-Chung Chen	153	Rob Polansky	305
Morgan Clark	113	Gerald J. Popek	183, 291
Douglas E. Comer	245	Liam R.E. Quin	231
Timothy W. Curry	267	Stephen Rago	113
Mark Daly	221	Marcus K. Ranum	37
Robert Dredge	221	Mary Pat Reeve	221
Dan Duchamp	161	David Ratner	183
Edward W. Felten	171	Peter L. Reiher	183, 291
Nathan Goodman	221	Steve Rozen	221
Jim Griffioen	197	M. Satyanarayanan	279
James D. Guyton	257	Michael F. Schwartz	257
James Gwertzman	305	Ken Shirriff	125
Jeffrey S. Haemer	1	Andrea H. Skarra	73
John Heidemann	183	Gregory Skinner	183
Dave Hitz	137	Carl Smith	137
Chet Juszczak	137	Peter Staubach	137
Brad Karp	305	Lincoln Stein	221
Geoff H. Kuenning	291	Shaowen Su	45
H. T. Kung	305	Diane Tang	305
Simon S. Lam	45	Thomas Y.C. Woo	45
Diane Lebel	137	Cheng-Zen Yang	153
Ian M. Leslie	209	Cliff Young	305
Kai Li	171	John Zao	305

A New Object-Oriented Programming Language: *sh*

Jeffrey S. Haemer
Canary Software, Inc.

Abstract

Many have frittered away their time on C++, while overlooking the new, POSIX.2-required, object-oriented language: *sh*. As will be clear from the enclosed code, the name may allude to the fact that the author would be embarrassed to have anyone find out about it.

This paper introduces a tiny, object-oriented programming system written entirely in POSIX-conforming shell scripts.

1. Overview

Object-oriented programming is currently all the rage [King89]. Though we normally use languages designed specifically for the task, they aren't always necessary. Here, we illustrate this point by doing object-oriented programming in the shell.

In what follows, object classes are shell scripts and objects are running processes. Methods are invoked by messages passed to objects through FIFOs (named pipes). The methods themselves are implemented as shell functions; function polymorphism is guaranteed because separate programs have separate name spaces. A class hierarchy is provided by the file system itself.

Sensible default actions are taken by objects when they're sent messages for which they lack explicitly defined methods. Debugging code can be added to objects on the fly, that is, after they've been created.

While the system is unconventional, only a toy, and downright slow, its implementation is straightforward and its use instructive. For example, figure 1 shows an implementation of the examples used in Roger Sessions' Summer, '93 USENIX Invited Talk [Sessions93].

Sessions' talk used application code size as one measure of the advantages of object-oriented programming. By that measure, and with the same examples, this system is better than C++. In fact, the core of the entire system, the two shell scripts *create* and *send*, total a little over 100 lines of code. If you don't find your favorite OOP feature, it may not be very hard to add it.

```
$ cat animalia
new animal pooh bugs

send pooh setName Pooh Bear
send pooh setFood Hunny
send bugs setName Bugs Bunny
send bugs setFood Carrots

for i in pooh bugs
do
    send $i getName
    send $i getFood
done

echo

new dog Snoopy
new littleDog Toto
new bigDog Lassie

for i in Snoopy Toto Lassie
do
    echo $i says
    send $i bark
    echo
done

destroy bugs pooh
destroy Snoopy Toto Lassie
```

```

$ ./animalia
My name is: Pooh Bear
My favorite food is: Hunny
My name is: Bugs Bunny
My favorite food is: Carrots

Snoopy says
Unknown Dog Noise

Toto says
woof woof
woof woof

Lassie says
WOOF WOOF
WOOF WOOF
WOOF WOOF
WOOF WOOF
WOOF WOOF

```

Figure 1. Animalia

2. Design

As a foundation, we begin by reviewing the three basic object-oriented features: encapsulation, function polymorphism, and inheritance. These requirements, plus sloth — an eagerness to let UNIX and the shell do as much of the job as possible — lead directly to most major design decisions.

- Encapsulation

All object-oriented systems provide data abstraction and encapsulation; they let programmers create and operate on objects that have user-defined types, while hiding all knowledge about how those operations and types are implemented. Programs are prevented from manipulating an object's internal data structures except through the methods that operate on those objects.

UNIX processes are attractive candidates for objects. Each running, UNIX process has its own name and address space; it's impossible to look at or tweak the insides of an already running process unless the process is running under a debugger.

- Objects and the object-class hierarchy

Object-oriented programming systems try to maximize code reuse by letting new data types inherit methods and the data structures they

operate on from their “parent” classes. This sort of inheritance produces a tree-structured class hierarchy. Implementing class hierarchies means picking a way to define trees.

UNIX has two ubiquitous tree structures that seem plausible choices, either of which might be worth barking up: the file system and the process tree. Deciding to make object classes conceptually distinct from objects, and making the latter simply instances of object classes, guides our choice. In this system, executing processes are objects; programs are their definitions, the object classes. The file system implements the class hierarchy. Although a process can be any executing program image, in this system all objects will be shell scripts.

- Polymorphism

A good object-oriented system lets the programmer extend the suite of object classes without changing existing software. Programs can operate on new kinds of data that get defined long after the code is written. The same operation may be implemented different ways for different kinds of data, but the invocation of the operation is identical. What separate compilation provides for functions, object-oriented programming provides for data.

For example, if each new object responds to *pass()*, there is no need to change base code from

```

    pass (X)
to
    if (object_type(X)==BILL)
        pass_bill(X);
    else if (object_type(X)==BUCK)
        pass_buck(X);
    else if (object_type(X)==GAS)
        ...

```

C++ and UNIX device drivers implement this sort of behavior by using tables of function pointers. *read()* is always *read()*, whatever the device, because the kernel figures out what routine to call based on the device that's being read from.

Many other object-oriented systems, like Smalltalk, implement polymorphism by passing messages between objects, which interpret the messages at runtime. Thus, the instruction

```
X pass
```

works whether X is a bill or a buck, because it

just sends a message. Both bill and buck understand the message "pass," but each implements the operation with a data-type-specific method.

Here, we take this latter approach. Class definitions are shell scripts. Methods are shell functions. Messages are just the names of the functions, sent as ASCII strings. Two different scripts are free to use identical names for completely different functions.

One side-effect of this approach is that messages sent to objects that are *not* names of functions are interpreted as other sorts of executable statements: built-ins and shell commands. On first blush, this seems like a horrible bug; in practice, and to my surprise, it feels like a feature.

3. Implementation

Each object is a simple, infinite loop:

forever
read message from FIFO
execute it as a command

The FIFOs are created in a pre-arranged spot in the file system and have names tied to the names of their corresponding objects. Messages are sent with the program *send*. The command

\$ send pooh setFood Hunny

just writes the message *setFood Hunny* to pooh's input channel /tmp/ ipc/pooh/in.

Object creation is trickier, but not by much. Each object class is a shell script, stored in a directory tree where the directory and subdirectory names are class and subclass names. Each directory contains one script, named *class*, that defines the methods for the class corresponding to that directory. A request to create an object of type *name* starts up a process like this:

use find to find directory name

*source all the class methods
from the root of the class tree
down to that definition*

*create a FIFO tagged to
the name of the object*

*loop forever, reading and executing messages
(as shown above).*

Taken together, *send* and *create* are currently only a little over 100 lines of shell code. (The *new* and *destroy* commands, used in the example in the first section, are just loops that call *create* and *send exit* for each of their arguments.)

A handful of interesting things fall out of implementing objects this way.

- Process manipulation commands can be used to handle objects. You can search for objects with *ps* and destroy them with *kill -9*.
- Every object understands normal shell commands. Not only can you see if an object is alive, but you can see if it's paying attention with commands like this:

```
$ send pooh date
Sun Jan 23 21:00:25 MST 1994
```

- You can kill objects with *exit*. This is enormously comforting. (Since I haven't gone to great lengths to make this system any more bullet-proof than it deserves to be, it's also enormously necessary.)
- You can add methods to objects on the fly. This sort of thing actually works:

```
$ send X 'zzazz() { echo foo }'
$ send X zzazz
foo
```

From time to time, this last feature has proven itself a useful debugging tool.

4. Real Code

Enough abstract chatter. Let's see some code.

4.1. send

send, shown in figure 2, sends messages to objects.

```
# send a message

case $1 in
  -d) msgtype=D
      shift ;;
  *)  msgtype=C ;;
esac

T_NAME=$1
shift

T_DIR=/tmp/ ipc/$T_NAME
```

```

T_IN=$T_DIR/in
T_OUT=$T_DIR/out
USAGE=\
"usage: $(basename $0) obj msg"

abort() { # print and bail
    echo $* 1>&2
    exit 1
}

test $# -gt 0 || abort $USAGE

test -d $T_DIR ||
abort $T_NAME: no such object

echo -e$msgtype "$*" > $T_IN
test $msgtype = "D" || cat $T_OUT

exit 0

```

Figure 2. send

All object I/O channels are in subdirectories of `/tmp/ ipc`. Each named object has a subdirectory that corresponds to its name, `T_DIR`, and all files associated with that object are within that directory. By default, each object has at least an input channel, `T_IN`, through which messages arrive, and an output channel, `T_OUT`, to which returns are sent.

The code shown above sets environment variables to point at the right channels, and then, after a brief sanity check, echoes its argument into the input channel and reads the objects response from the output channel.

There are at least two restrictions of this design. First, the name space is global to the system; only one object on the entire system can be called “foo” at any given time. Second, there’s no provision for tying returns to the messages that elicited them; if two objects send messages to a third at nearly the same time, there isn’t any way to guarantee that the return value one of the senders retrieves corresponds to the message it sent. A more sophisticated implementation might nest object directories as subdirectories under the directories of the objects that created them, and use a more sophisticated messaging scheme to provide a virtual circuit between the messenger and the messagee.

Even after accepting these limitations, at least two problems require immediate solution.

The first of these is the deadlock that arises when object A sends a message to object B and object B, or some object further down the line, sends a message to object A before object B has replied to A’s original message. In these cases, object A cannot read the incoming message because it is blocked reading B’s output channel. The general case is a general problem, but in some cases object A doesn’t really need B’s answer, and can go on to listen for incoming messages as soon as it dispatches a message to B. For just such cases, `send` accepts a flag, `-d`, that means “don’t wait for an answer.” `Send` prepends a ‘D’ to such “datagrams,” and replies are neither expected nor supplied.

The second problem is trickier. In the absence of special arrangements, an open of a FIFO for writing will only complete when that FIFO has an available reader. Consider, then, what happens when object A sends a message to itself, using a command like

```
echo $msg > /tmp/ ipc/A/in.
```

The `echo` will block, awaiting a reader, preventing A from ever executing the read that would move `echo` past the block.

The current implementation side-steps this problem by providing each object with a built-in version of `send`. Whenever an object notices that it is sending a message to itself, it executes the message directly instead of trying to write the message to its own input channel. (See figure 7.) An alternative to this would be putting `echo` in the background, but that would use up process slots, a resource that this system already strains. Another alternative might be writing substitutes for `read` and `echo`.

4.2. create

More complex than `send` is `create`, shown in figure 3.

```

# create a new object

O_DIR=/tmp/ ipc/$2
O_IN=$O_DIR/in
O_OUT=$O_DIR/out

O_BIN=${O_BIN:-$HOME/obj/bin}
O_CLASS=$1
O_NAME=$2
O_PATH=/bin:/usr/bin:$O_BIN
O_ROOTS=${O_ROOTS:-$HOME/obj/objs}
export O_BIN O_CLASS O_NAME

```

```

export O_PATH O_ROOTS

USAGE=\
"usage: $(basename $0) class obj"

abort() {
    echo $* 1>&2
    exit 1
}

test $# -eq 2 || abort $USAGE

# cleanliness is next to godliness
cleanup() {
    trap "" 0 1 2 3 15
    rm -rf $O_DIR
    exit 0
}

event_loop() {
    trap "cleanup" 0 1 2 3 15
    while read pkt < $O_IN
    do
        type=${pkt%% *}
        msg=${pkt#[A-Z] }

        if test $type = "D"
        then
            PATH=$O_PATH eval $msg
        else
            PATH=$O_PATH eval $msg \
                >$O_OUT
        fi
        # hack around BSDI timing bug
        sleep 1
    done
}

get_obj_chain() {

    # find object and superclasses
    IFS=:
    set $O_ROOTS
    IFS=' '
    for i
    do
        test -f $i/class || continue
        obj_root=$(
            find $i \
                -type d \
                -name $O_CLASS \
                -print
        )
        test -n $obj_root && break
    done
}

done
test -z $obj_root && abort $USAGE

# now set up paths
d=$obj_root
O_PATH=$O_PATH:$d
O_DEFS=$d
while test "$d" != "$i"
do
    d=${d%/*}
    O_DEFS=$d" $O_DEFS"
    O_PATH=$O_PATH:$d
done

unset d i
}

# create message channels
make_channels() {
    test -d $O_DIR &&
    abort "Duplicate $O_NAME"
    mkdir -p $O_DIR ||
    abort "Can't make $O_DIR"
    mkfifo $O_IN ||
    abort "Can't make $O_IN"
    mkfifo $O_OUT ||
    abort "Can't make $O_OUT"
}

# build the object from definitions
mkobj() {
    for d in $O_DEFS
    do
        . $d/class 2>/dev/null
    done
}

get_obj_chain
make_channels
mkobj
event_loop &

```

Figure 3. create

Following initialization and sanity checks, **create** makes four function calls to create an object. The first, *get_obj_chain*, sets the variable **O_DEFS** to a list of directory names, starting at the root object directory, that end in the directory that defines the class. For the class *littleDog*, from our earlier example, **O_DEFS** would be set to *objs objs/animal objs/animal/dog objs/animal/dog/littleDog*

Next, *make_channels* creates the input and output channels used by *send*.

Third, *mkobj* visits the directories in *O_DEFS*, reading class definitions. Because of the order in which *get_obj_chain* sets up *O_DEFS*, methods defined in subclasses supplement or override those defined in parent classes.

Finally, *event_loop* loops infinitely, reading messages and writing responses on the pair of message queues set up by *make_channels*. If the message is the name of a function call — a method — that function is invoked. Otherwise, *eval* looks for a shell built-in or a UNIX command to execute.

A disadvantage of the approach sketched above is that there isn't an easy way to say "use my parent's definition of this method"; when an object definition overrides a method defined by a parent, that parental method becomes completely unavailable.

In an earlier version of this code, *event_loop* looked like this:

```
event_loop() {
    trap "cleanup" 0 1 2 3 15
    while read msg < $O_ICHAN
    do
        eval $msg > $O_OCHAN
    done
    cleanup
}
```

In the version shown in figure 3, *get_obj_chain* stores the path to the directory that contains the object definition in *O_PATH*, for later use in creative ways, including prefixing it to the path used by *eval*. Having that path available makes it possible to back up through the class hierarchy searching for a parental method. I've experimented with this, but the trick isn't entirely satisfactory; a more sophisticated implementation should find a more interesting way to use *O_PATH* or *O_DEFS* to gain access to parental-class methods.

Like *send*, which has a single, global namespace for objects, *create* uses a global name space for object classes. The system will not support two different definitions of class "dog". On the other hand, users can point at their own object-class definitions by setting *O_ROOTS*, even invocation by invocation.

Another limitation of this system is that it is restricted to single inheritance; each class has one and only one parent class — that of its parent directory. Although links might make it possible to provide an interesting way to implement and explore

multiple inheritance, everyone knows that multiple inheritance is a bad idea [Cargill91].

4.3. new and destroy

Returning now to the example, we can show *new* (figure 4) and *destroy* (figure 5).

```
# create a set of objects

USAGE=\
"new class obj [obj ...]"
abort() {
    echo $* 1>&2
    exit 1
}

test $# -ge 2 ||
abort "$USAGE"

class=$1
shift

for i
do
    create $class $i
done
```

Figure 4. new

```
# destroy a set of objects

USAGE=\
"destroy obj [obj ...]"
abort() {
    echo $* 1>&2
    exit 1
}

test $# -ge 1 ||
abort "$USAGE"

for i
do
    send $i destroy
done
```

Figure 5. destroy

As advertised earlier, each of these is a simple loop. The earliest version of *destroy* was even

simpler:

```
for i
do
    send $i exit
done
```

This works because each object inherits the methods understood by the base class — the shell augmented by a few basic methods. The current version calls `destroy` instead, which lets each object define its own destructor. (The base class defines a simple default `destroy`, shown in the next section.)

One alternative to explicit calls to `destroy` for each object would be to make `new` keep track of objects it has created and let `destroy` destroy everything.

Because of encapsulation, the easiest implementation would incorporate `new` as a method in a more sophisticated base class. An odd side-effect of this is that classes could redefine `new`.

4.4. hop: a simple class

Having constructed the infrastructure, let's look at a simple class definition (figure 6).

```
$ cat objs/hop
# class hop

hop() {
    if test "$*" = "on pop"
    then
        echo -n "Stop! "
        echo "You must not hop on pop."
    else
        echo "hippity hop"
    fi
    return 0
}
$ create hop X
$ send X hop
hippity hop
$ send X hop on pop
Stop! You must not hop on pop.
$ send Xfoo
$ send X exit
$ send X hop
X: no such object
```

Figure 6. Class hop

The entire class definition is a single method: `hop`.

Although this is an elementary example, it illustrates a few interesting points:

- (1) Defining methods is easy; it doesn't require a lot of special syntax.
- (2) Class definitions are small. While code size is not the only measure of the quality of a programming language — else we would all program in APL — code size strongly affects maintenance and debugging efforts; bug frequency per line appears to be roughly constant across languages [Brooks75]. Less code, fewer bugs.

As an illustration, Sessions contrasts the code to make a dog bark in C:

```
void printDog(dog *thisDog,
              int dogType)
{
    printf("\n%s says\n",
           getName((dog *) thisDog));
    switch dogType {
        case DOG:
            dogBark(
                (dog *) thisDog);
            break;
        case LITTLEDOG:
            littleDogBark(
                (littledog *) thisDog);
            break;
        case BIGDOG:
            bigDogBark(
                (bigdog *) thisDog);
            break;
    }
}
```

with the code to do the same job in C++:

```
void printDog(dog *thisDog)
{
    printf("\n%s says\n",
           thisDog->getName());
    thisDog->bark();
}
```

Here's the same code in the shell:

```
echo $thisDog says
send $thisDog bark
```

- (3) We've chosen to ignore nonsense requests. The consequences of changing that decision can be explored by toying with `event_loop` in

create

- (4) Methods can have arguments.

This example becomes even more interesting when we notice that we can invoke methods that aren't defined by the class. As figure 7 shows, methods defined by parent classes (in this case, the class defined in directory *objs*) are inherited by their subclasses

```
$ create hop X
$ send X self
X
$ send X class
hop
$ cat objs/class
# fundamental methods

abort() { # print and bail
    echo $* 1>&2
    exit 1
}

class() {
    echo $O_CLASS
}

debug() {
    echo $O_NAME: $*
}

defs() {
    for d in $O_DEFS
    do
        cat $d/class 2>/dev/null
    done
}

_destroy() {
    test $# -eq 0 && exit 0
    test $0 = "self" && exit 0

    for i
    do
        send -d $i destroy
    done
}

destroy() {
    _destroy $*
}

self() {
```

```
    echo $O_NAME
}

send() { # send a message

    case $1 in
        -d) msgtype=D;
            shift ;;
        *) msgtype=C ;;
    esac

    T_NAME=$1
    shift
    T_DIR=/tmp/ ipc/$T_NAME
    T_IN=$T_DIR/in
    T_OUT=$T_DIR/out

    USAGE="usage: send obj msg"
    test $# -gt 0 || abort $USAGE

    if test "$T_NAME" = "$O_NAME" ||
       test "$T_NAME" = "self"
    then
        PATH=$O_PATH eval $*
        return 0
    fi

    test -d $T_DIR ||
    abort $T_NAME: no such object

    echo -e $msgtype "$*" > $T_IN
    test $msgtype = "D" || cat $T_OUT
}
```

Figure 7. The base class

Most of the methods defined in *objs/class* are simple utility methods. The motivation for the one long method, *send*, was given in section 4.1.

4.5. animals

The code for our original animal example shows inheritance in practice (figure 8).

```
$ cat objs/animal
# base animals methods

name=$O_NAME
food="Unknown food."
```

```

setName() {
    name=$*
}
getName() {
    echo My name is: $name
}
setFood() {
    food=$*
}
getFood() {
    echo My favorite food is: $food
}
$ cat objs/animal/dog
# dog: all bark, no bite

bark() {
    echo Unknown Dog Noise
}
$ cat objs/animal/dog/littleDog
# a little dog

bark() {
    echo woof woof
    echo woof woof
}

$ cat objs/animal/dog/bigDog
# a BIG DOG

bark() {
    for i in 0 1 2 3 4
    do
        echo WOOF WOOF
    done
}

```

Figure 8. Animal Objects

Here, the class *animal* defines methods for setting and getting an animal's name and favorite food; the subclass *dog* adds a way to make the animal bark, and sub-sub-classes for little and big dogs replace that method with ones that generate size-appropriate noises.

5. Applications

Sir, a woman's preaching is like a dog's walking on his hinder legs. It is not done well; but you are surprised to find it done at all.

Boswell's *Life of Johnson*, vol 1, p 428, 31 July 1763

"Cute idea," you say, "but is this good for implementing real applications?" Probably not. Still, it seems worth sniffing around to see what sorts of things besides barking dogs might be interesting to implement with it.

5.1. Starting small ...

When I was soliciting suggestions for interesting applications to implement, Doug Pintar, of Aztec Engineering, laconically suggested *emacs*. While an *emacs* implementation might not fit within the page-limit length imposed by this conference, I can include a more contained, but logically equivalent, application. Appendix A shows the code for a Turing machine. The text below, sketches the implementation of each of the classes. The example, taken from the nearest automata theory text to hand [Manna78], recognizes strings of the form

$$a^n b^n$$

5.1.1. Turing machine The machine itself is an object that creates a tape object and five nodes. After initializing all the objects, loading the tape with an input string and the nodes with their transition tables, it starts up by telling the first node to go, and then awaits an announcement of success or failure from some node down the line. When the announcement arrives, the machine writes the result as output; destroys the nodes it has created; and exits.

5.1.2. Tape

The tape itself is trivial. Input data are stored as a string, and there are a handful of methods to move along the tape and to read or write at the current position. (We've tried to avoid mixed-case disease, which seems endemic to object-oriented programmers, but *Read* requires an initial, upper-case 'R' because *read* is reserved by the shell. *Write* is just following suit.)

The position is just a numeric index into the string, maintained using the POSIX shell's built-in arithmetic facilities.

5.1.3. Node Nodes, too, are objects. When called on, a node reads the current cell on the tape and looks up the entry for the character it reads in a dictionary of transitions that it creates and maintains (another object, described in the next section). Transitions are a triple containing a character to write into the current cell, a direction to move, and a new node to call. The current node writes the prescribed character into the

cell, moves either left or right, and then calls on another node (possibly itself) to handle the next cell. If the dictionary reveals that the node has reached a decision to accept or reject the input string, then instead of passing control to a new node, the current node sends the message *accept* or *reject* to the original Turing machine.

The absence of returns and lack of a single, centralized transition table lend a palpably unstructured aura to the process.

Although this implementation uses a colon-separated array to store the three pieces of information associated with each possible input character and teases them apart with **cut**, performance could be improved somewhat by using the shell's prefix- and suffix-shaving operators — `$(PARAMETER%%expression)` and friends — to do the parsing without recourse to subprocesses.

5.1.4. Dictionary A dictionary stores its entries as shell variables whose names are constructed on-the-fly from the words being defined. The command

```
$ send $DICTIONARY define and dumb
```

turns into the assignment

```
def_and=dumb
```

In a better world, the shell might have arrays (indeed, the Korn shell does), but POSIX shells aren't required to have them, and this work-around is good enough for our example.

5.2. ... then getting smaller.

Exploiting the ability of objects to learn new methods at run-time, we can also create a simpler but more tantalizing application. The code shown in figure 9 shows a method that sends itself to another object: a virus.

```
$ cat oneline
# put everything on one line

tr -s '\n\t' '[ * ]'
$ infect()
> {
>   send $1 \"$(typeset -f infect|oneline)\""
> }
$ new null X
$ infect X
$ send X typeset -f infect
infect ()
```

```
{
  send $1 "$(typeset -f infect | oneline)"
}
$ new null Y
$ send X infect Y
$ send Y typeset -f infect
infect ()
{
  send $1 "$(typeset -f infect | oneline)"
}
```

Figure 9. A Simple Virus

The script **oneline** is a work-around for two implementation problems. First, with the code shown here any methods learned at runtime must fit on a single line. Second, shell quoting conventions make it annoyingly difficult to fit the **tr** command inside the function itself.

(We confess to having resorted to occasional non-standard shell extensions to avoid other lengthy circumlocutions, particularly **echo -e** which interprets many of the usual shell escape characters like '`\n`', and **typeset**. All these shell scripts run under **bash**, a publicly available POSIX-conforming shell, and the extensions are those provided by **bash**. Other POSIX shells provide analogous extensions.)

A more sophisticated *infect* would go out and hunt for other objects to infect. A more sophisticated *create* routine would permit multi-line messages.

5.3. Summary Neither of these applications is particularly long (or useful), but each illustrates the capability and extensibility of this relatively simple system, and the power and flexibility of the shell as a programming language.

As a parting note, we observe that, the two applications can be used in consort: despite its simplicity, limitations, and implementation dependencies, the virus shown above can be used to infect the Turing machine described above to give it a cold in its nodes.

6. send Paper exit

This is hardly a complete system. On the other hand, it's so simple that an average undergraduate who's already familiar with UNIX at the shell level should be able to play with objects without first having to

wrap his mind around a conventional OOPS like C++ or Smalltalk. A really good undergraduate should be able to enhance it in interesting ways without a course in compiler theory. I've suggested several such enhancements in this paper.

What's more, even though the exercise seems akin to making a sow's ear out of a silk purse, it illustrates that the shell has more power than many people give it credit for. That said, I'll raise anew a question posed by Steve Johnson at the Winter '94 USENIX conference: "Will object-oriented programming replace the shell?" Johnson intended the question to be rhetorical, but I harbor the suspicion that object-oriented shells, and other shells that break from the conventional-programming-language model, are fruitful areas of research [Budd89]. Mashey showed that creating a shell that was a real programming language was exactly the right idea, and that people would use a well-designed shell early and often. [Mashey76]. Given that, I'm surprised that nearly all widely available shells today still use C, ALGOL, or pocket-teller machines as their models.

(A notable exception is Doug Gwyn's "Adventure Shell." Though not a wild success as a programmer's shell, it has spawned, after a trip through a maze of twisty passages, the development of MOOs.)

Personally, I've long wanted to run "the spread-shell" but I haven't any idea what such a thing would do. If you write one, please send it to me.

Acknowledgements

My thanks to Dave Taenzer for patient explanations about objects, Jim Oldroyd and Doug Pintar for humorous discussions about object-oriented applications, Dick Dunn for working me into a lather about them, and Rob Pike for an apposite quote. Thanks also to Mike Karels for fixing FIFOs in BSDI on the spot at a POSIX meeting.

References

[Budd89] Tim Budd. "The design of an object-oriented command interpreter," *Software Practice and Experience*, 19(1), pp. 35-51 (January 1989).

[Brooks75] Fredrick P. Brooks, Jr.. *The Mythical Man-Month: Essays on Software Engineering*, Addison Wesley, Reading, Massachusetts. pp. 93-94 (1975).

[Cargill91] T. A. Cargill, "The Case against multiple

inheritance in C++," *Computing Systems*, Vol 4(1), pp. 69-82 (1991).

[Johnson94] Steve Johnson, "Objecting to Objects" USENIX Winter Conference Invited Talks Submitted Notes, San Francisco, January 1994, pp. 41-61.

[King89] Roger King, "My Cat is Object-Oriented," in *Object-Oriented Languages, Applications, and Databases*, W. Kim & F. Lochovsky, eds., Addison-Wesley, New York. (1989).

[Manna78] Zohar Manna, *Mathematical Theory of Computation*, McGraw-Hill, New York. pp. 22-23 (1978).

[Mashey76] J. R. Mashey, "Using a command language as a high-level programming language." *Proceedings of the Second International Conference on Software Engineering*, San Francisco, California. pp. 177-181 (October 1976).

[Sessions93] Roger Sessions, "An Introduction to Object-Oriented Programming and C++" USENIX Summer Conference Invited Talks Submitted Notes, Cincinnati, June 1993, pp. 29-38.

Biography

Jeffrey S. Haemer is an independent consultant in Boulder, Colorado. He works, writes, and speaks on the interrelated topics of internationalization, POSIX, open systems, software portability, and porting. Dr. Haemer has given UNIX programming tutorials since 1988 and is a frequently featured speaker at such well-attended industry forums as Expo Kuwait, USENIX, and the Romanian Open Systems Exposition. He currently serves as the USENIX organizational representative to the POSIX effort.

Appendix A: A Turing Machine

```
# turing machine
#   recognizes a^n b^n

MACHINE=$O_NAME
TAPE=${MACHINE}_T
export MACHINE TAPE

destroy() {
    #debug destroy $*
    _destroy $TAPE s1 s2 s3 s4 s5
    exit
```

```

}

accept() {
    echo ACCEPT!
    destroy
}

reject() {
    echo REJECT!
    destroy
}

new tape $TAPE
new node s1 s2 s3 s4 s5

send $TAPE load aabb

# Hard-wire the nodes.
# It'd be nicer to have this
# load from a file.

send s1 transition a A:right:s2
send s1 transition _ X:right:accept

send s2 transition B B:right:s2
send s2 transition a a:right:s2
send s2 transition b B:left:s3

send s3 transition B B:left:s3
send s3 transition a a:left:s4
send s3 transition A A:right:s5

send s4 transition a a:left:s4
send s4 transition A A:right:s1

send s5 transition B B:right:s5
send s5 transition _ X:right:accept

send -d s1 goto

```

Figure A1. Turing Machine

```

# Turing machine tape

unset S
typeset -i n
typeset -i j

right() {
    let n=n+1
    return 0
}

```

```

left() {
    if test $n -le 1
    then
        echo HALT
        return 1
    else
        let n=n-1
        return 0
    fi
}

load() {
    S=$1
    let n=1
    return 0
}

print() {
    echo $S
    let j=n
    while let j=j-1
    do
        echo -n ' '
    done
    echo '^'
    return 0
}

Write() {
    let left_neighbor=n-1
    let right_neighbor=n+1
    left=$(echo $S |
        cut -c -$left_neighbor)
    right=$(echo $S |
        cut -c $right_neighbor-)
    S=${left}${S}${right}
    return 0
}

Read() {
    if test $n -gt ${#S}
    then
        echo '_' > $O_OUT
    else
        echo $S |
        cut -c $n >$O_OUT
    fi
    return 0
}

```

Figure A2. Turing Machine Tape

```
# Turing machine node
```

```

XITIONS=dict_$O_NAME
new dict $XITIONS

transition() {
    send $XITIONS define $*
    return 0
}

destroy() {
    send -d $XITIONS destroy $*
    _destroy $*
}

goto() {
    SYMBOL=$(send $TAPE Read)
    ACTION=$(
        send $XITIONS lookup $SYMBOL
    )

    debug $SYMBOL, $ACTION

    if test -z "$ACTION"
    then
        send -d $MACHINE reject
        return 0
    fi

    OUT_CHAR=$(echo $ACTION |
        cut -f 1 -d:)
    DIRECTION=$(echo $ACTION |
        cut -f 2 -d:)
    NEXT_STATE=$(echo $ACTION |
        cut -f 3 -d:)

    if test $NEXT_STATE = "accept"
    then
        send -d $MACHINE accept
        return 0
    fi

    send $TAPE Write $OUT_CHAR
    send $TAPE $DIRECTION

    send -d $NEXT_STATE goto
    return 0
}

```

Figure A3. Turing Machine Node

```

# Small dictionary

dictionary() {
    set | sed -n 's/^def_//p'

```

```

        return 0
    }

define() {
    eval def_$1="$2"
    return 0
}

lookup() {
    eval echo $"def_$1"
    return 0
}

```

Figure A4. Simple Dictionary

The Old Man and the C

*Evan Adams
Sun Microsystems*

Abstract

"You can't teach an old dog new tricks" goes the old proverb. This is a story about a pack of old dogs (C programmers) and their odyssey of trying to learn new tricks (C++ programming).

C++ is a large, complex language which can easily be abused, but also includes many features to help programmers more quickly write higher quality code. The TeamWare group consciously decided which C++ features to use and, just as importantly, which features not to use. We also incrementally adopted those features we chose to use. This resulted in a successful C++ experience.

1.0 Introduction

This paper describes the experience of a group of C programmers adopting C++ for a new project. It is written from the viewpoint of C programmers and describes our expectations, surprises, pleasures, disappointments and trials and tribulations. It is intended for C programmers that may be considering a journey into the realm of C++. It is not intended to be a critique or evaluation of C++ as a programming language.

The TeamWare project consisted of 8 very experienced C programmers, ranging from 4 to 13 years of industrial C experience. In the spring of 1991 we decided to implement the TeamWare project in C++. It was hoped that using C++ would lead to more code sharing, more cleanly structured code and improved internal interfaces. No one within the project had any significant experience with C++, although two members of the group had some object oriented experience.

TeamWare is a set of command line and GUI tools built from several common libraries. The libraries are

provided by the TeamWare group for use by the TeamWare applications; they are not provided for more general use.

TeamWare is a code management product that encourages parallel development and is built on top of SCCS. A user makes a copy (bringover) of an SCCS hierarchy thus creating a personal hierarchy. In this hierarchy the user makes and tests changes. These changes are then integrated (putback) into the original hierarchy. If the integration hierarchy contains changes which are not in the user's hierarchy, then TeamWare detects that there have been parallel changes and refuses the integration. Therefore, users must incorporate changes in the integration hierarchy into their own hierarchy before integrating. TeamWare also includes the filemerge utility, a graphical three-way differences program allowing users to merge parallel changes. TeamWare tracks both source file changes (SCCS deltas) and file renames.

1.1 Which Way To Go?

In the beginning, the group was faced with two paths. The first path, marketed by Nike, was labeled "Just Do It" and appealed to our impulsive nature. The second path, labeled "Crawl Before You Walk", appealed to our logical selves. The Just Do It path called for each of us to decide for ourselves which features of the language to use and how to apply them. The Crawl Before You Walk path called for the group to use new features of C++ only as it became apparent that they added value over our more well understood C techniques.

1.2 Getting Started

We began by taking a C++ course taught by Hank Shiffman of SunPro Marketing and offered through

Sun U. and buying a handful of books [Ellis, Stroustrup 1990], [Eckel 1990], and [Dewhurst, Stark 1989]. We found the Annotated Reference Manual to be somewhat daunting for the average programmer. As a language reference manual it is intended more for compiler writers and people interested in a very precise language definition. The other two books explain how to use the language. Dewhurst and Stark's book is much more concise and was, therefore, the first reference. Eckel's book was used when Dewhurst and Stark's was inadequate. Many C++ books have been published since the spring of 1991 so there may well be better options available now.

Initially, some of us felt that C++ would be pretty easy to pick up. After all, wasn't it just C with a bit more stuff? Hank's class convinced us otherwise. We left the class feeling that there was a lot to this language, some parts we liked, some parts we didn't and much that we didn't fully understand. For example, we left the class with the clear message - "Stay away from multiple inheritance".¹

We chose the Crawl Before You Walk path and started with very modest goals; we would use classes instead of structures, constructors and destructors and member functions. We felt certain that our future would include inheritance and virtual member functions, but we did not feel ready for them yet.

2.0 Features We Used

2.1 Required Function Prototypes

A *function prototype* is a function declaration containing the function's return type and the types of all its arguments. Function prototypes allow the compiler to do strong type checking as the compiler ensures that a function is always called with parameters of the appropriate types. C++ requires a function prototype for every function that is called.

Initially, we ported some utility code from a previous project. It had been written in Kernighan & Ritchie (K&R) C. The first task was to change all the function declarations from the K&R C style to the C++ style, and to declare the function prototypes in the header files. This was tedious work, but quickly demonstrated the power of requiring accurate function prototypes. We kept compiling the files until the compiler no longer complained, knowing that, only then, did the uses match the definitions. In the long run, we found required function prototypes to be the

single biggest advantage of C++ over K&R C or even ANSI C.²

Our early C++ days were very frustrating with respect to the C++ error messages. We found the error messages to be obscure and not terribly informative. Each of us had experiences of spending several hours trying to figure out what the compiler was telling us.³ Coming from C programmers this is no small statement. Over time this problem went away. We concluded that the C++ error messages are probably not much worse than the C compiler's, but it took us a while to gain the same familiarity with them that we have with the C compiler's.

2.2 Classes

Classes are the essence of C++'s object model. They are like C structures with the addition of constructors and destructors, public and private fields, member functions and the ability for one class to inherit from another. A class generally consists of the data needed to present a certain concept. The member functions are routines that operate on that data and form the interface to the class.

2.3 Constructors and Destructors

Constructors are routines that initialize newly allocated objects and *destructors* are routines that clean up before an object is de-allocated. An object is created by having the new operator allocate memory, and then the constructor initializes the memory. Likewise, an object is destroyed by having its destructor called to clean up the object (such as closing open file descriptors), and then having the delete operator de-allocate the memory. The new and delete operators replace traditional malloc() and free() usage.

We became fans of constructors and destructors. Much of our C code had followed the same principles by providing one routine which would allocate an instance of the type and initialize its fields, and by providing a second routine which would de-allocate the appropriate member fields and the instance of the

1. We did.

2. ANSI C has function prototypes but they are not required. ANSI C and C++ interpret the declaration void func() differently. In ANSI C, it is a function with no argument type checking, while in C++ it is a function with no arguments. The ANSI C equivalent is void func(void).

3. A favorite was a message suggesting that "perhaps a ; was missing" which was frequently generated when there was an extra semicolon.

type itself. We were pleased to have language support for what we had been doing by hand.

One aspect of constructors we found annoying is that they must be kept very, very simple because it is awkward to have a constructor return an error value, such as failure to open a file. We kept our constructors simple and then added a member function to do things which might fail. However, this separates the complete construction of an object into two, possibly separated, pieces. It is possible to end up with a partially constructed object. In one case, we passed into the constructor the address of an error variable so the constructor could return an error.

2.4 Function Overloading

Function overloading allows you to have more than one function with the same name as long as those functions take different types of arguments. In C, the use of a function named *foo* maps to one and only one function defining *foo*. With function overloading, this is no longer true. The reader must take into account the arguments passed to *foo* to correctly map *foo* to its implementation.

As old C programmers we left the C++ class with an uneasy feeling about function overloading. We were fairly convinced that use of function overloading would be confusing and not prove to be worthwhile.

However, constructors encouraged us to use function overloading. We often found it advantageous to provide more than one constructor for a class. Frequently, we would provide a very bare-bones constructor which initialized all the member fields to default values along with additional constructors which did more and more sophisticated initialization. We found this type of function overloading to be very natural and useful. Overloaded constructors probably represented about 90% of all our overloaded functions. The remaining overloaded functions tended to be ones which accepted different numbers of arguments. The ones accepting fewer arguments would supply default values for the missing arguments and call the one accepting the most arguments. Default arguments could have been used instead but, since they made us nervous, we chose to make the defaulting explicit.

2.5 Member Functions

Member functions are a suite of functions associated with a class. They can be viewed as providing the interface to the class, that is, the set of operations which can be applied to instances of the class. A member function is called via a pointer to an object or

an actual object. Each member function is implicitly passed a *this* pointer, which is a pointer to the object through which the call was made. Inside a member function the scoping rules change. The member fields and functions can be referenced directly, it is not necessary to use the *this* pointer.

Member functions were a big hit. At this point, we were using objects and member functions, but no inheritance. Without inheritance, member functions are primarily syntactic sugar, but one we took a liking to. One of the larger weaknesses of C is that there is a single global name space for all functions. In C++, each class provides a separate name space for its member functions. This results in member functions being given less verbose and more descriptive names, resulting in more readable code.

Initially, we found directly referencing member fields and functions rather disconcerting as we were referencing names which were not declared to be either local or global. Furthermore, we sometimes declared a parameter with the same name as a member field. The compiler did not complain about this either and the scoping rules result in the parameter hiding the member field. The latter can be a serious problem and was the source of several very subtle bugs.

Classes and member functions cause C++ to have more name spaces than C, so the naming conventions used in C programs often turn out to be inadequate for C++. It would be advantageous to use a naming convention which syntactically separates fields from parameters and variables. We never completely came to grips with this problem.

2.6 Public, Private, and Protected Fields

Member fields in a class can be either *public*, *private* or *protected*. Public fields can be referenced from any object (.) or object pointer (->); private fields can be referenced only from within its class's member functions and its friends; protected fields are the same as private unless inheritance is used.

Some kinds of fields are really private to a class. These keep track of the internal state of an object and users of the class have no need to either read or write them. Other fields are of interest to the users of a class. Making these private requires that functions be provided to get and set the field. We called these accessor functions.

We did not discuss group-wide conventions for using public, private and protected fields, and consequently two very different styles emerged. The people writing the libraries dabbled with private

members, but did not find that they added much value and eventually took to declaring all new members public. The people writing the GUI applications made much more significant use of private member fields and their corresponding accessor functions. When debugging, they found it useful to be able to set a breakpoint in a set routine and catch all situations where a given field was being set.

In hindsight, we believe we would have made much greater use of private fields if we had been providing a public API. Private fields give the implementors of an API much greater control over the interface by preventing arbitrary access. In our case, we were providing an API to ourselves and we did not find this level of interface control necessary.

2.7 Inline Functions

Inline functions have their bodies expanded at each call site. They eliminate the function call and return overhead in exchange for duplicate copies of their bodies.

A frequent objection C programmers have to accessor functions is - "Why should I have to make a function call just to get the value of a field? This will be too expensive!". Inlined functions provide a very nice solution to this problem. They allow the implementor of a class to tightly control its interface without needlessly sacrificing performance. When we used private data, the corresponding get routine was almost always inlined. Set routines were frequently inlined as well.

A common question regarding inlined functions is - "What size function should be inlined?". In answering this question you must consider the performance gain resulting from removing the function call overhead, versus a possible increase in code size. Someone suggested to our group a rule of thumb we think makes sense - do not inline any function which contains control structures.⁴

2.8 Public, Private, and Protected Member Functions

Public, private and protected apply to a class's member functions as well as its fields. Just as for fields, they define a member function's scope. We found we used private member functions less often than private fields.

4. Strict interpretation of this rule would prevent simple if statements, however, these can be done with the ?: operator.

As with private fields, we believe we would have made greater use of private member functions if we had been providing a public API as they separate the interface from the implementation.

2.9 Inheritance and Virtual Functions

Some of the early code we ported included a list package. This took us on our first journey into inheritance and virtual functions. *Inheritance* provides for building one class (a derived class) from another (a base class). The derived class becomes a superset of the base class and has all the base class's functionality along with any new functionality provided by the derived class. *Virtual functions* allow the derived class to modify the behavior of the base class. If the base class contains a virtual function and the derived class provides a function with the same name and arguments, then the derived class's function supersedes the base class's.

Inheritance and virtual functions encourage implementing generic functionality in a base class, and then specializing that functionality in each of the derived classes. A list package is a good example. Much of the implementation of a list package is generic and applies to all lists regardless of the elements they contain. However, some of the implementation is specific to each type of list. Printing the elements of a list is an example; walking the list to visit each element is generic while the actual printing of an element is specific to the element type.

Many C programmers are initially confused by the semantics of inheritance and virtual functions. They frequently have trouble determining whether a given call will invoke the base class's function or the derived class's function. Say you have a base class with a non-virtual member function then you can call this function via an object derived from the base class. If this non-virtual function in turn calls a virtual function, does it call the base class's function or the derived class's function? The answer is obvious to experienced C++ programmers⁵, however it is often confusing to C programmers first learning C++.

We found it much easier to understand these semantics by thinking about the implementation. First, the C++ compiler will try to resolve as many function references as it can at compile time. Any reference to a non-virtual function is resolved at compile time. Any reference to a virtual function cannot be resolved at compile time. Secondly, a class's virtual functions are put into a table of virtual function pointers and this

5. It will call the derived class's function.

table is associated with every instance of that class. In the earlier example, when a non-virtual function is called from a derived object, a pointer to the derived object is passed into the function (the `this` pointer). The virtual function is then found in the `this` pointer's virtual function table. Therefore, the derived object's function gets called.

2.10 List Package

We had two goals for our list package. First, to preserve the ability to have lists of things which were unaware they were in a list. Second, to have *typed lists*, that is, a list of `ints`, a list of `char *`s, a list of pointers to `class foos`, etc., rather than generic lists. The first couple of attempts at converting the list package were feeble. The interfaces were clumsy, there were many friend declarations, and usage was awkward. The third iteration settled down into what we felt was a pretty reasonable interface and by this time all the friend declarations had disappeared. We also concentrated on making it easy to create lists of new types.

It has become apparent that object oriented languages do not deal well with implementing generic container classes. In C++ this is the motivation behind templates. However, we were using `cfront` 2.0 which predated templates.

An example of the generic container problem is shown by trying to copy a list. Copying a list is something the base list class should do as it understands the implementation of lists. A virtual function should be used to copy an element of a list as this allows each derived list to provide its own copy function. So far so good, but what type does the base class's copy routine return? That is the dilemma. The only type it knows about is the base class, yet this is the wrong type to return since it is copying a typed list. We found it necessary to have the base class's copy routine return a pointer to the base class and then have each derived class also supply a copy routine. The derived class's copy routine then calls the base class's copy routine and casts the return value to a pointer to the derived class's type.

This same problem occurs for a few other functions, and applies to all derived lists. Therefore, we wrote a macro which, given the derived list's type, generates all the necessary functions.

Likewise, there are several virtual functions which manipulate the elements of a list. The elements are stored as opaque types so each of these functions needs to cast the opaque type to the appropriate element type.

We wrote a second macro to generate these functions.

We believe that templates would have led to a cleaner solution to these problems but have not yet had a chance to try them. In the end, we were pleased with the list package (and its cousin the hash package). Our libraries contain nine different types of lists. Several members of the team commented that creating new list types was very easy and beneficial.

2.11 More Inheritance

We applied inheritance and virtual functions in several other places as well. They are very powerful concepts. It takes a bit of effort to become comfortable with them but their use can generate significant rewards. We found that using classes, inheritance, and virtual functions resulted in greater code sharing. Obviously, this level of code sharing was possible using C but it takes much more discipline. With C++, the language provided support for these concepts, making it much easier to achieve code sharing.

For those familiar with TeamWare, its `bringover` and `putback` commands do very similar things yet they differ in a few areas. They are implemented with a base class called a `Transaction` and derived `Bringover` and `Putback` classes. The derived classes do things like argument parsing and implement the differences between the `bringover` and `putback` commands while the bulk of the work is done in the `Transaction` class.

It was common, when a member of the group first encountered this implementation, for them to ask - "how do I tell if I'm in a `bringover` or `putback` command? Where is the global variable to test?". The answer would be - "There is no global variable. An object knows which one it is so, if it you are doing something unique to one of the commands, then it should be done in a virtual function.". This was a new way of thinking.

Sometimes we would create a class not expecting it to become a base class only to discover later on that we needed to derive another class from it. We then faced the question - should all the base class's functions be made virtual, or should only those functions which our derived classes replace be made virtual? We did a little of both with no obvious results favoring either technique. We feel this issue comes back to whether or not you are providing a public API. If so, then you probably want to make your classes very flexible and allow derived classes to replace many of the functions. This implies that most, if not all, public member functions should be virtual.

Otherwise, our experience indicates that it really doesn't matter.

2.12 Pure Virtual Functions and Abstract Base Classes

A *pure virtual function* is a virtual function in a base class for which no actual function is defined. A pure virtual function is declared by putting = 0 at the end of its declaration.⁶ If a class contains a pure virtual function, then it is an *abstract base class*. The significance is that the compiler will not allow you to have any instances of an abstract base class.

Abstract base classes should be used when the base class is so generic that it is not useful just by itself. Only when some key functionality is supplied by a derived class does it become useful. Our list package was an example of an abstract base class. With typed lists, an instance of the base list class is not meaningful. Without pure virtual functions, the base class's implementation of these functions would probably consist of printing a nasty message and then exiting.

2.13 Operator Overloading

Operator Overloading is the ability to redefine the basic C++ operators. TeamWare's applications did not lend themselves to needing operator overloading. We only overloaded the new and delete operators for some classes so that we could impose our own memory management. This was very useful and allowed us to elegantly gain significant performance improvements.

General operator overloading seems like a feature that is likely to be abused. During the C++ class, Hank warned us to never overload an operator to do something entirely different than the operator's normal semantics. This is very reasonable advice. If you have an object and doing things like adding two of them together makes sense, then operator overloading may be the way to go. However, we would recommend that it be used with caution.

2.14 Calling C Routines From C++

It is common to need to call C routines from C++ as many libraries, such as libc, contain routines written in C. At first glance, this wouldn't appear to pose any

6. The syntax is abysmal. An = followed by 0 is a pure virtual function. An = followed by anything else is an error.

problems. However, since C++ allows function overloading, it is forced to perform *name-mangling* on function names. That is, if you have three different functions named *foo*, then C++ has to invent a unique name for each of them. External routines written in C will not have mangled names, so C++ allows you to indicate that a given function is a C routine and that its name should not be mangled. This is done by preceding a declaration with *extern "C"*⁷.

The *extern "C"* declarations provide a nice mechanism and, when needed, are absolutely crucial.

2.15 Calling C++ Routines from C

Calling C++ routines from C is an entirely different matter. There are two ways to call external C++ routines from C. You can either deduce the routine's mangled name and call it, or you can define the global C++ routine to be *extern "C"* and defeat the name mangling.

It is more challenging to call C++ member functions from C because you don't have objects in C. Say you have a C++ class and a corresponding structure in C. To call a member function, you would need to write a wrapper routine in C++ which takes the structure as a parameter, converts it to an object and then calls the appropriate member function. The wrapper routine must also be prepared to convert any return values.

We called global C++ routines from C and did so by deducing the mangled names⁸. This was the wrong way to do it. We were unaware of the *extern "C"* technique until Hank pointed it out while reviewing this paper. We never tried to call C++ member functions directly from C.

Having mangled names in our C sources leaves us at the mercy of the C++ compiler. There is no guarantee that all C++ compiler's will mangle names in the same way or that a given C++ compiler will always use the same technique. This creates both portability and maintenance problems. The *extern "C"* approach is the civilized technique.

2.16 Comments

C++ introduces a new syntax for comments, // is a comment until the next newline⁹.

7. This is strange syntax. "C" is not a keyword as much as a key-string-literal.

8. We did this with nm.

9. What does this have to do with adding objects to C?

This is another topic that we never discussed amongst the group. Consequently, some people chose to use // and others /* */. Furthermore, we used gxv++ and it generated code with // comments. Ultimately we ended up with some files using //, some using /* */ and, worse yet, some using both.

There is clearly no right or wrong answer here. However, it would have been preferable if we had all used the same style.

2.17 `set_new_handler()`

The new operator uses `malloc()` to allocate a new object. If it is unable to allocate memory, then it returns a NULL pointer. The C++ library routine `set_new_handler()` allows you to register a function to be called when the new operator fails.

`set_new_handler()` provides a nice way to intercept `malloc()` failures within the built-in new operator. The alternative is to check the return value of every call to the new operator. We used `set_new_handler()` to register one routine for command line programs and a different routine for GUI programs. The command line routine printed an error message and exited and the GUI routine displayed a pre-allocated notice and then exited after the notice was dismissed.

3.0 Features We Chose Not to Use

Until now, this paper has described the features of C++ we used and our experiences with those features. This section describes the features we chose not to use and why.

3.1 Multiple Inheritance and Virtual Base Classes

Multiple inheritance allows a derived class to inherit from more than one base class. Each base class can also be the product of multiple inheritance, creating an inheritance DAG. A class derived via multiple inheritance exports an interface which is the union of the interfaces exported by all its base classes. The inheritance DAG can include the same base class more than once. In this situation, *virtual base classes* control whether or not the derived class has just one or many instances of the base class.

When considering inheritance the *is-a* versus *has-a* relationship is crucial. If a derived class is one of another class, then inheritance is proper; however, if the derived class has one of a another class, then aggregation is proper. For example, an editor window

would derive from a generic window class because, it is a window. An editor window would also have a font, but it would not inherit from the font class. Rather, it would contain an instance of the font class or a pointer to an instance. For multiple inheritance to be proper, a derived class needs to be one of several other classes. We never encountered this situation.

The C++ books we had were little help either. Despite Waldo's [1991] protests to the contrary, we agreed with Cargill [1991] that the multiple inheritance examples in the books were contrived and could have been more cleanly expressed with aggregation.

We found single inheritance easy to understand and use and extremely powerful. However, we found multiple inheritance to be very complicated and confusing. With multiple inheritance, the reader of a class definition must assimilate the entire inheritance DAG complete with virtual and non-virtual base classes to understand the class. Multiple inheritance creates an awkward ambiguity when two base classes have a member with the same name and the name resolution rules in a complicated inheritance DAG are extremely difficult.

More than any other feature in C++, multiple inheritance appears to be a large wart on the language.

3.2 Reference Parameters

Reference parameters allow you to pass parameters by reference rather than by value. There are some situations in which reference parameters are highly desirable, namely operator overloading, where they clean up an otherwise very ugly syntactical situation. There are other cases where they can cause confusion such as for base types. Reference parameters are an attempt to clean up some of the notation associated with pointers in C. In C, if you have a structure and want to pass a pointer to it as a parameter you must take the structure's address with the & operator. References let the compiler do this for you.

Linton [1993] contends that reference parameters aid in storage management. Typically, a local object is passed as a reference parameter. Therefore, the receiving function is promising not to store the parameter's address in a global data structure.

More fundamentally, a C++ programmer must decide if their basic programming model is pointer based or object based, that is, do they have pointers to objects or just objects. This is complicated by the fact that objects can be declared globally and locally, yet the new operator returns a pointer to an object, not the

object itself. Reference parameters assume that you are object based, not pointer based, however, this makes it more awkward to use the new operator. Most real applications use the new operator because most applications need to dynamically allocate objects.

Since we were all experienced C programmers, we were comfortable with the notation associated with pointers. We had also developed a programming model in our heads which reference parameters change. For example, in C, parameters cannot be changed in a way which affects the calling function. Early in our development someone used a reference for an `int` parameter. Another person was reading the code and thought he had found a bug because the `int` was being passed to a routine and the next line clearly assumed the `int`'s value had changed. We all knew an `int` passed by value could not be changed in the caller so this must be a bug. However, since the caller declared a reference parameter, the compiler was actually passing in the address of the `int`, so the parameter was being modified by the call. We avoided references because of that incident, because our code made heavy use of the new operator, and because we did not use operator overloading. There are some tricks which old dogs can't learn.

3.3 Friends

Friends allows you to specify that a function or an entire class can access a class's private member fields and functions.

We observed that friends are generally used when interfaces are not cleanly defined. They can be thought of as the *casts of classes*, that is, a way to circumvent the language's built-in safeguards. We were not successful in completely avoiding friends, but we strongly suspect that the two places we used them signify flaws in our interfaces. We discourage the use of friends.

3.4 I/O Streams

C++ provides an *I/O Streams* package as a replacement for C's `stdio` package. `Stdout` is represented by the `cout` object and output is done with the overloaded `<<` operator; `stdin` is represented by the `cin` object and input is done with the overloaded `>>` operator.

We took an initial dislike to this package. We were perfectly comfortable with `printf`, we were told that I/O Streams have somewhat worse performance than the `stdio` package, and we did not care for the syntax. Actually, this package violates one of the

axioms for operator overloading that Hank told us about - "Never overload an operator for something other than its normal purpose". This package overloads the shift operators for input and output. Deciding to not use C++'s I/O streams was probably one of the quicker decisions we made.

3.5 Operator Overloading

As mentioned above, we overloaded the `new` and `delete` operators for some classes. Otherwise, we did not use this feature. We felt that operator overloading could be a very seductive feature, as you can somewhat create your own language. This temptation should be avoided. There are probably some very good situations for operator overloading, but care should be taken to ensure they are not used gratuitously.

3.6 Default Arguments

Default arguments allow a function call to be missing its trailing arguments and they will receive default values. Default arguments are essentially a short-hand for writing several overloaded functions.

One person in our group used a few default arguments and liked them. The rest of us never felt they were necessary and in similar situations wrote the overloaded functions.

3.7 Local Declarations Anywhere

C requires that local variables be declared at the beginning of statement blocks. C++ lets you declare variables anywhere.¹⁰ For example, if an `int` is used just in a `for` loop, then the `for` loop can be written:

```
for (int i = 0; ...
```

We found very little practical value to this feature.

4.0 Features We Didn't Think About

C++ is a large, complex language. The following are a set of features that we really didn't think about and therefore did not use.¹¹

4.1 Global Objects

Global objects are instances of classes with a global scope. They must be initialized via a constructor

10. What does this have to do with adding objects to C?

11. Since we didn't think about them, this list is probably incomplete.

before `main()` is called. C++ sets up the executable this way, however, the order of invocation of the constructors is not defined. So, if one global object's initialization depends upon another there are potential ordering problems.

We did not have any global objects, in part because we used a pointer based programming model, so we never encountered the ordering problem. However, another group in SunPro did encounter this problem and warned us about it.

4.2 Copy Constructors

Copy constructors are used when an object is copied. For example, if a class has a `char *` field where each instance has its own copy of the string, then, when this object is copied, the `char *` field should be duplicated rather than just copying the pointer. Copy constructors provide this mechanism.

This is another situation which the pointer based programming model seems to avoid.

4.3 Static Member Fields

Static member fields are fields which are shared across all instances of an object. They can be considered global data which is scoped by a class, that is, they can be accessed only through an instance of a class.

4.4 Static Member Functions

Static member functions are member functions which are not passed a `this` pointer. Like static member fields, they can be considered global functions which are scoped by a class.

4.5 Const

`Const` is new to ANSI C and to C++. Unfortunately it is somewhat different in the two languages. We were coming from a K&R C world without const.

We never gave consts much thought. Looking back we do not recall any situations in which consts would have saved us. Still, they seem like a reasonable feature, especially in interfaces to class libraries. If an interface rigorously uses consts in its declarations, then users can know which parameters may be modified by which routines. This looks like another feature which is more valuable when producing a public API.

5.0 Conclusions

The TeamWare group felt that using C++ was an advantage and that using C++ helped reduce development time and increase the quality of our code. We found the single largest advantage to be required function prototypes. We converted two existing C programs¹² to C++ and in both cases function prototypes found one or two latent errors. Required function prototypes eliminate an entire class of errors which occur in C and give programmers much more confidence. However, function prototypes alone are not sufficient reason to use C++ as an ANSI C compiler which enforced the use of prototypes would be a more sensible choice.

We were very pleased with C++'s object model. This includes classes, constructors, destructors, single inheritance, member functions, virtual functions and abstract base classes. We found these features of the language easy to adopt and easy to understand. C++'s object model encourages the good programming practices of modularization, well defined interfaces and code sharing. The object model provides valuable functionality which sets C++ apart from ANSI C.

One significant disappointment with C++ is that it does not separate a class's interface from its implementation. Abstract base classes are nice as they provide an interface specification. However, if the class also has private member functions, they must be defined along with the public ones. If a private member function is added to a class, its interface has not changed but, in the world of make, the header file has changed and all files including that header will be recompiled.

Unfortunately, the object model includes multiple inheritance and, consequently, virtual base classes. Most examples of multiple inheritance found in C++ books are contrived and should be replaced with aggregation. They do not provide good models to follow. Multiple inheritance adds complexity to a program. It should only be used if the is-a relationship is satisfied for each inheritance and if the result is simpler than aggregation. We believe that groups which follow this advice will rarely, if ever, use multiple inheritance.

Second only to multiple inheritance is the decision regarding a pointer based programming model versus an object based programming model, or whether or not to use reference parameters. C programmers tend to gravitate towards the pointer based programming model as it is familiar to them. While we favored the

12. filemerge and make.

pointer based model, it is more important that a decision be made and that the decision be applied consistently throughout the project. In the case of public APIs it might be appropriate to supply both pointer and reference interfaces as this allows users of a library to choose their own style.

We chose to use a fairly small subset of C++ as we found it had an over abundance of features. Unfortunately, the ANSI C++ committee is not done adding to the language. Exceptions and templates were added after we started our project. The language is still growing and looks like it will include run-time type information (RTTI), name spaces and additional cast operators. We fear that as C++ grows it becomes less usable. This may result in C trying to adopt a useful subset of C++ features. The ANSI C committee has voted to reconvene itself and has received a proposal to add classes with single inheritance [Jervis 1993]. While we welcome these features into C, we fear the war between C and C++ that may result.

We are convinced that incrementally adopting C++ features and making conscious decisions about which features to use and how to use them was the right thing to do. If anything, we should have discussed more thoroughly some of the features of the language, such as private and public members and comments.

One final anecdote regarding the earlier story where someone used an `int` reference parameter. In discussing this paper, the programmer's comment was - "Well, the feature was in the language so I figured I should use it.". It is our belief that this is not a sufficient criteria for using a feature of C++. A feature should be used only when it can be demonstrated to be of benefit. A mountain is climbed "because it is there". The same should not hold true for C++ features. Their mere existence is not justification for use.

6.0 Summary

Below is a table of C++ features along with our assessments:

Feature We Used	Comments
Function Prototypes	Most valuable feature
Objects	Second best feature
Classes	Well done
Constructors/Destructors	Good programming practice
Member Functions	Good name space scoping
Single Inheritance	Promotes code sharing
Virtual Functions	Powerful; promotes code sharing
Pure Virtual Functions	Ugly syntax; valuable
Abstract Base Classes	Ugly syntax; valuable
Function Overloading	Used mostly for constructors
Inlined Functions	Very nice
Calling C from C++	Strange syntax; nice feature
Comments	Why?
set_new_handler()	Very convenient
Private or Protected Fields	Did not use effectively
Private or Protected Functions	Did not use effectively
Operator Overloading	Used only for new and delete
Calling C++ from C	Very awkward

Features We Chose Not to Use	Comments
Multiple Inheritance	Too complicated
Virtual Base Classes	Too complicated
Reference Parameters	Old dogs can't learn new tricks
Friends	Indicative of problems
I/O Streams	Saw no advantage over stdio
Default Arguments	Used function overloading instead
Local Declarations Anywhere	Why?

Features We Didn't Think About	Comments
Global Objects	Did not need them
Copy Constructors	Used a pointer-based model
Static Member Fields	Provides tighter scoping
Static Member Functions	Provides tighter scoping
Const	Good for public APIs

Acknowledgments

I would thank the TeamWare development team of Jules Damji, Jill Foley, Claeton Giordano, Lewie Knapp, Daniel O'leary, Marla Parker, Mark Sabiers and our manager John Treacy.

References

- T. Cargill, Controversy: The Case Against Multiple Inheritance in C++, *Computing Systems*, 4(1); Winter 1991.
- S. C. Dewhurst and K. T. Stark, *Programming in C++*, Englewood Cliffs, NJ; Prentice Hall, 1989.
- B. Eckel, *Using C++*, Berkeley, CA; McGraw-Hill, 1990.
- M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Reading, MA; Addison-Wesley, 1990.
- R. Jervis, *Classes in C*, Working paper for ANSI/ISO WG14/N298 X3J11/93-044, 1993.
- M. A. Linton, private communication, 1993.
- J. Waldo, Controversy: The Case Against Multiple Inheritance in C++, *Computing Systems*, 4(2); Spring 1991.

Biography

Evan Adams. Has been at Sun Microsystems for over 11 years working on compilers, debuggers (wrote the original dbxtool) and other programming tools. Spent 4 1/2 years at Amdahl Corporation working on UTS (Unix on IBM mainframes). Graduated with a BS in Computer Science from Oregon State University in 1978. Email address: evan@eng.sun.com.

Key Management in an Encrypting File System

*Matt Blaze
AT&T Bell Laboratories*

Abstract

As distributed computing systems grow in size, complexity and variety of application, the problem of protecting sensitive data from unauthorized disclosure and tampering becomes increasingly important. Cryptographic techniques can play an important role in protecting communication links and file data, since access to data can be limited to those who hold the proper key. In the case of file data, however, the routine use of encryption facilities often places the organizational requirements of information security in opposition to those of information management. Since strong encryption implies that only the holders of the cryptographic key have access to the cleartext data, an organization may be denied the use of its own critical business records if the key used to encrypt these records becomes unavailable (e.g., through the accidental death of the key holder).

This paper describes a system, based on cryptographic "smartcards," for the temporary "escrow" of file encryption keys for critical files in a cryptographic file system. Unlike conventional escrow schemes, this system is bilaterally auditable, in that the holder of an escrowed key can verify that, in fact, he or she holds the key to a particular directory and the owner of the key can verify, when the escrow period is ended, that the escrow agent has neither used the key nor can use it in the future. We describe a new algorithm, based on the DES cipher, for the on-line encryption of file data in a secure and efficient manner that is suitable for use in a smartcard.

1. Introduction

Modern distributed computing systems, for all their virtues, make it difficult to limit reliably access to sensitive data. Networks often unselectively broadcast data to far-reaching and unpredictable places,

remote login facilities create new opportunities for trespassers and distributed file systems often assume that all machines to which they provide service are trustworthy and reliable. To reduce these risks, cryptographic techniques make it possible to limit data access while still taking advantage of untrustworthy networks and services. Modern workstations can encrypt in software at close to network speeds [4][5]. Data encryption attempts to ensure that only those who possess the correct decryption key can obtain the cleartext data.

Most commercial applications of encryption techniques protect communication links (and related services such as electronic mail). When communication endpoints are under the control of a single entity, or trust a common authority, the management of cryptographic keys is a conceptually straightforward matter. Keys can be assigned and changed as often as desired, the main problem being to ensure that both sender and receiver agree as to the current keys and that keys are discarded when no longer in use. Should sender and receiver get "out of sync" with the keys, the problem becomes immediately apparent because communication fails. Ensuring access by third parties in the event that keys are lost or unavailable is rarely an issue.* Public key techniques [3][10] make communication key management easier, allowing two parties to establish a secure channel without prior arrangement.

*The law enforcement community argues that it may be an exception; widespread use of encryption techniques may impede police wiretap investigations [2]. The ethical, legal, social and technical implications of law enforcement access to cryptographic communication are presently the subjects of intense public debate in the United States and are (fortunately) outside the scope of this paper.

Cryptography can also be used to protect file data, although there are relatively few tools for this purpose in widespread use. Most file encryption takes place at the application level, with tools such as the Unix **crypt** command or with special encrypting applications (e.g., "vi -x"). File encryption can also take place at a lower level, as a basic service of the file system [1][9][13].

Regardless of where encryption takes place, key management for encrypted files is a fundamentally different problem from that in cryptographic communication. In a secure communication system, keys must be distributed and synchronized *geographically*. Keys often serve the dual purpose of authenticating identity as well as protecting against eavesdroppers. The architecture for distributing communication keys is closely tied to the trust relationships within the system, and practical key distribution protocols (such as those employed by the Kerberos system[12]) must be carefully engineered to balance reliability, security and performance.

In a file system, on the other hand, there is usually little need to distribute keys geographically; most protected files are encrypted and decrypted at the same locations (and by the same users). Authentication of identity is a less serious issue, with access implicitly controlled through knowledge of the key itself, although cryptographic techniques can also be used to detect unauthorized tampering with file data. File systems still present a significant, if differently formulated, key management problem, however, in that keys can be said to be distributed *temporally*. The corresponding keys must be available at both encryption and decryption time. File encryption keys have much longer lifetimes than their communication counterparts. If a key is lost or unavailable, the files encrypted with it are rendered useless. This condition may not be detected until it is too late. The key distribution center and public key cryptographic protocols developed for geographically distributed communication systems do not have direct analogues that can be readily applied to temporal file key management.

Arguably, it is because of difficulties associated with key management that sensitive files are rarely encrypted in practice even when encryption tools are available. This is especially true in critical business environments where ensuring the availability of data to authorized users is at least as important as ensuring its unavailability to everyone else. Sometimes, files are protected with weak ciphers, such that the encrypted data can be recovered with the application of sufficient computing resources. A toolkit ("Crypt

Breaker's Workbench") is available in Internet archives for the purpose of decrypting files enciphered with the Unix **crypt** program. Needless to say, since these tools are also available to the adversary, encryption with weak ciphers is of questionable value in the first place.

In the context of organizational information systems, cryptographic file protection presents several problems not addressed by traditional (communication-oriented) key management schemes. These problems are not only technical (e.g., providing mechanisms for ensuring that keys are available when and where authorized) but also managerial and social (balancing secrecy and privacy against emergency access requirements). Carefully controlled key management services with explicit, auditable trust relationships that are integrated into the underlying file system security architecture can help reconcile these often conflicting goals.

2. Key Escrow

Hence the problem: strong file encryption is often necessary to protect privacy while availability requirements sometimes dictate the need for a "back door" for emergency access. We use as our model the common problem of ensuring continued access to critical business files even after the only employees who know the keys to those files leave the organization. One approach adapts the procedures used for controlling physical locks and keys to file encryption keys and provides a central key distribution ("locksmith") service. Any time a user requires an encryption key, it is generated by a central service, which also keeps a copy for emergency access.

In practice, however, the central locksmith model adapts poorly to large-scale file encryption key management. The central service must be unconditionally trusted by all who obtain keys from it. No further controls preclude or audit access by those with access to the key database. (Note that this is not the case with locksmiths who manage physical keys — use of a key requires access to the lock, which may itself be controlled by independent security mechanisms and which can be changed if the locksmith's office is compromised. In the case of file keys, on the other hand, once a copy of the key database has leaked, all files with keys in the database must be considered compromised forever.) Furthermore, a central service can quickly become a service bottleneck or worse, a single point of failure or attack. The key service is an "online" part of the key creation process and users cannot create new

keys if the service is unavailable. Finally, the problem of securing communication between the user and the key center introduces all the problems of communication key management in addition to the existing problem of file key management.

An alternative approach reverses the relationship and provides a controlled mechanism for users to deposit copies of their keys for emergency use as needed. The keys for crucial files could thereby be "escrowed" with a trusted caretaker who would reveal them only when certain conditions are met, such as when encrypted business data are required after the death of the legitimate key holder. Conceptually, keys might be delivered within sealed "envelopes." When a set of files is no longer critical, the envelope containing its keys could be returned to its originator, who could verify the integrity of the seal and destroy the keys, preventing future access to outdated, but still private, data. The "escrow-deposit" approach has the benefit of allowing the key holder to generate keys in the usual manner, without direct "online" interaction with a third party. There is no central service bottleneck, since the escrow agent is not directly involved in the creation of new keys. Envelopes containing escrowed keys can be delivered to the escrow agent at any time and any inability to deliver the keys to the agent need not preclude their use by the key holder.

Unfortunately, this is difficult to do in practice. The simplest procedure has the key holder write down the key, place it in a sealed envelope, and leave it with a trusted caretaker. This is vulnerable to mistakes, however, since there is no inherent mechanism to ensure that the escrowed key is the same as the real one. The security of the scheme also depends entirely on the honesty of the caretaker and the tamper-resistance of the envelope. An electronic analogue to the sealed envelope can be implemented by encrypting the key with a "caretaker" key, perhaps using public key techniques. If this is done automatically as part of key generation, the problems associated with transcription mistakes are avoided, but the scheme still depends entirely on the caretaker's honesty (and even more so without the sealed envelope). If no single caretaker can be trusted, the key could be multiply encrypted with more than one caretaker's key, split among several escrow agents (in the manner of the US Escrowed Encryption Standard) or encrypted using a group-oriented public key protocol.

Both the manual and encrypted key escrow schemes suffer from a fundamental problem, however. After an escrow agent "opens" the key and learns its value, no further controls on its use are

possible. Anyone who learns the keys can use them at any time in the future without detection. Electronic escrow is particularly hard to revoke or audit, since it is difficult to ensure that all copies of the keys have been destroyed when the escrow period ends even if the keys have never actually been used (consider backups or illicit copies of the escrow data).

Under these schemes, key escrow is an "all or nothing" proposition, with no mechanism to guarantee, in any formal sense, that the caretaker is doing his or her job honestly. It is not obvious how to implement key escrow schemes that offer stronger protection against abuse without relying on elaborate physical access controls or special purpose hardware.

Cryptographic smartcards can be used to implement more carefully controlled and fully revocable file system key escrow. Smartcards have several properties that lend themselves to use as a controlled store for escrowed keys. These cards are designed to be sufficiently tamper-resistant to allow their use in financial applications, have a controlled-access non-volatile memory, can run general purpose software and include built-in cryptographic and random number generation capabilities.

3. Smartcard-Based Key Escrow in a Cryptographic File System

The shortcomings of entirely software-based key escrow schemes arise out of the inability to control the use of the key once it has been revealed to the escrow agent. Thus the problem is to guarantee the escrow agent use of the key without actually revealing what it is. While this may appear to involve impossibly contradictory requirements, most commercial smartcards can be adapted to serve exactly this purpose.

We propose a system in which an "escrow smartcard" can be created along with each file encryption key. This card is provided to a designated third party (the "escrow agent") who is authorized to use the key under some well-defined set of circumstances. If emergency access is required the card can decrypt files without revealing what the key is, acting as a self-contained decryption engine for ciphertext sent to it by the escrow agent. Any time the card decrypts data it also records that fact in its secure storage. Later, when the escrow period is terminated or when an audit is to be performed, the user can query the card to determine whether the escrow agent has used it. This section describes the design and implementation of a smartcard-based key escrow scheme for CFS, a file encryption system for Unix.

CFS is a cryptographic file system interface for Unix-like systems; it allows the user to associate cryptographic keys with directories. It runs entirely on the client workstation. No modification to the underlying file system (or file server) is required, and file contents as well as some meta-data (file names) are cryptographically protected. Backups and other such routine administrative services can take place in the normal manner and without the encryption keys. Details on CFS can be found in [1].

Basically, CFS provides a mechanism to associate "real" directories (on other file systems) that contain encrypted data with temporary "virtual" names through which users can read and write cleartext. These virtual names appear in a separate namespace under the CFS mount point, which is usually called `/crypt`. Users create encrypted directories on regular file systems (e.g., in their home directories) using the `cmkdir` command, which creates the directory and assigns to it a cryptographic "passphrase" that will be used to encrypt its contents. To use an encrypted directory, it must be "attached" to CFS using the `cattach` command, which asks for the passphrase and installs an association between the "real" directory and a name under `/crypt`. Cleartext is read and written under the virtual directory in `/crypt`, but the files are stored in encrypted form (with encrypted names) in the real directory. When the directory is not in use, the association is removed with the `cdetach` command, which deletes the cleartext virtual directory under `/crypt`. When CFS is run on a client workstation, the cleartext data (and the cryptographic key passphrase) are never stored on a disk or sent over a network, even when the real directory is located on a remote file server. The system is implemented as a user-level NFS[11] server. The basic flow of data in CFS is shown in Figure 1.

Key escrow is implemented for CFS as an option to escrow the key when the encrypted directory is created with `cmkdir`. When keys are initially assigned and whenever escrowed access is required, the machine running CFS must have a smartcard reader-writer attached. (In day-to-day user operation on encrypted files, no smartcard reader is required.) The smartcard has a small store of secure memory, the ability to run simple programs securely and a secret-key cryptographic engine compatible with that of the host file system. Ideally, the card could have a real-time calendar and the ability to schedule execution at some future date, although the cards we use (the AT&T smartcard) do not have these capabilities. We call the user who created the files the "owner" and the caretakers of the escrowed keys the "escrow

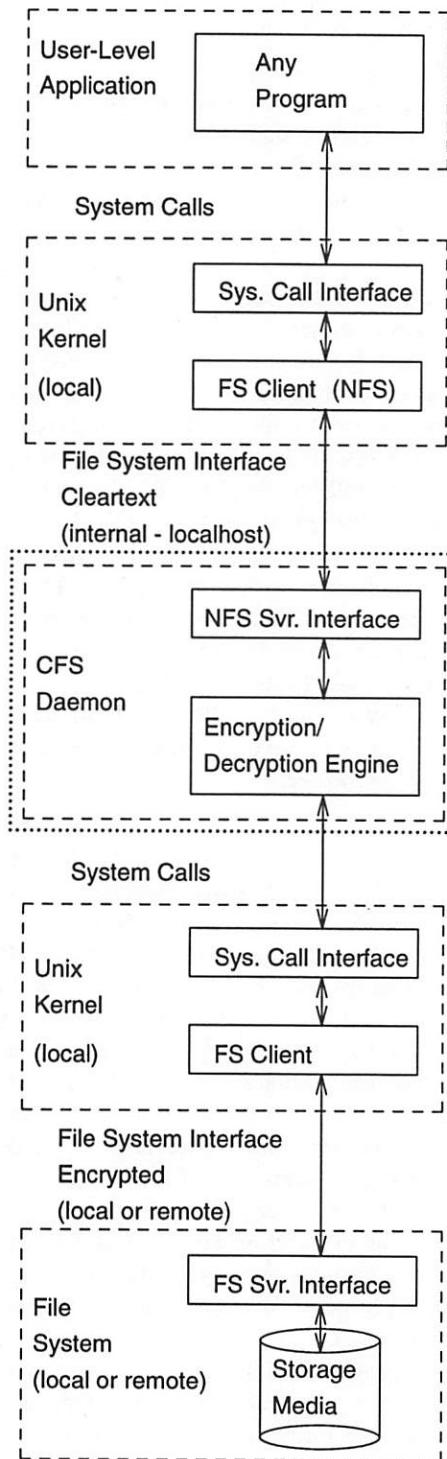


Figure 1 – Data Flow in Standard CFS System

agents." The techniques described here could be applied to any file encryption system and are not limited to CFS.

At the time keys are assigned (e.g., with the CFS cmkdir command), the smartcard is initialized with three sets of cryptographic keys. The first key set, the "file system key," is used for actual file data encryption, and consists, in CFS, of two 56 bit DES[6] keys derived from a user-selected secret "passphrase." The file key is also used to hash a known plaintext string that is stored in the host file system in the "check file." The second key, the "audit key," is used to post-audit the card at escrow revocation time and will be explained in more detail below. The audit key is also stored in a file on the host computer (encrypted under the file keys). The last key, the "escrow key," is used to encrypt the file system keys stored on the card. It must also be provided to the escrow agent (perhaps via public key techniques, and perhaps split among several agents, but this key is not essential to the security of the protocol). Ordinarily, the escrow key is derived from a second passphrase entered by the owner. The encrypted file keys and audit key are maintained in secure storage on the card and cannot be easily "reverse engineered" from the card. All smartcard initialization takes place in CFS through a modified version of the cmkdir command.

Once keys are assigned, the smartcard is turned over to an escrow agent for safekeeping and the escrow key passphrase revealed to the escrow agent. (The escrow agent who holds the card need not be the same agent who knows the escrow key). If the smartcard has a calendar and the ability to schedule future execution, the escrow data on the cards could be configured to automatically self-destruct after a set period. If needed, duplicate cards, with new escrow and audit keys, can be created by the owner (using the file passphrase) at any time.

In normal CFS operation, the file system keys are derived from the user passphrase on the trusted host computer when the owner issues the "cattach" command for an encrypted directory; the smartcard is not involved. Regular user operation requires only the standard version of CFS (without any escrow software). The check file assures that the entered phrase is valid and that wildly incorrect decrypted file names and contents are not returned to the file system.

The smartcard itself is used to perform three operations. The first operation, "pre-audit," simply verifies to the escrow agent that the keys on the card correspond to those used to encrypt the actual file

system. In this mode of operation, the escrow agent sends the contents of the check file (in the escrowed file system) and the escrow key to the smartcard, which provides a "yes" or "no" answer based on the decrypted file keys. (The owner could "cheat" and provide a "dummy" check file; we discuss this below.) The escrowed keys do not leave the card.

In "escrow access" operation, the smartcard decrypts files for the escrow agents. The agents supply the escrow key; if it is supplied correctly, the card decrypts the file system keys and increments a counter in its secure store. Thereafter, for the remainder of the session, the card will use the decrypted file keys to decrypt file data sent to it. If the card has a real time clock, it could also maintain two time stamps for the first and most recent times the escrow key was used. Again, the keys never leave the card; the card acts as a wholly self-contained decryption engine. Once the card is removed, its state is reset and the escrow key must be supplied again to enable further decryption. Escrow access in CFS takes place through a modified CFS file system daemon in which the crypto engine is replaced with calls to the smartcard interface. Additional support tools supply the escrow key to the smartcard. Note that the card interface is part of the data path for all decrypted data. The data flow is shown in Figure 2.

The last mode of operation, "post-audit," is used when the escrow period is ended and the card is returned to the owner. The card reports the number of times the escrow keys were used. If the card has the capability to store this data it could also report the first and last access times and number of bytes decrypted under escrow (again, our cards do not). To help protect against card forgery and to safeguard against the return of a fake card by the escrow agent, the owner can challenge the card to perform encryptions under the audit key. The audit key is decrypted on the host computer with the owner passphrase; by comparing the results of a random challenge with the result of a decryption performed locally, the owner can verify that the card that was returned is the same one that was originally escrowed. Post audit is performed in CFS with an additional user tool.

3.1. File Encryption Scheme

One of the lessons learned from the design of CFS is that the problem of encrypting files on-line in a file system is somewhat different from other kinds of encryption problems. No single standard encryption mode[7] has all the properties required for file system use; further compounding the problem are concerns

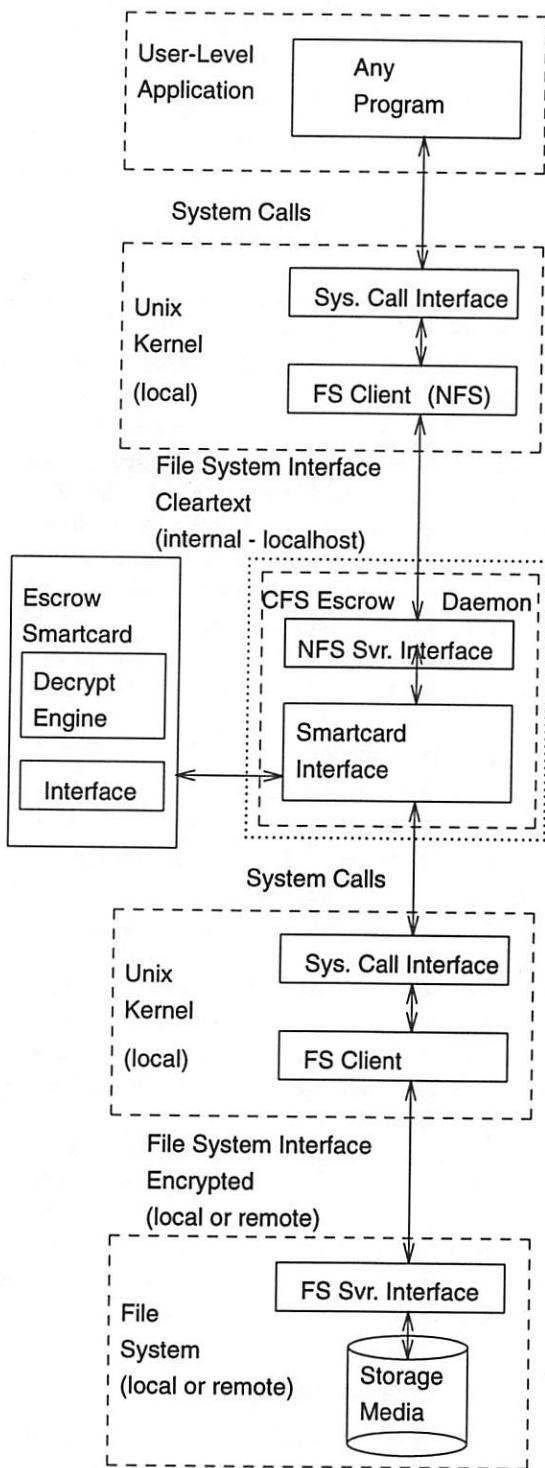


Figure 2 – Data Flow in CFS Escrow Agent System

that the 56 bits of key used by the DES cipher are vulnerable to exhaustive search attack[14].

CFS uses a combination of DES "codebook" and pre-computable "stream" cipher modes to approximate the strength of multiple iterations of DES with the runtime latency of only a single iteration of DES. This scheme has the resistance to structural analysis of a chaining cipher but allows random read and write access in constant time.

The encryption scheme relies on the ability to trade off space (in the precomputation of the streams) for time. To accommodate the key escrow system, we modified the CFS encryption scheme to allow "lazy evaluation" on the smartcard without the large memory requirements of the precomputed stream. We believe this scheme to be equivalent to 3-DES under currently known practical attacks. The new CFS cipher is as follows:

Recall that keys in CFS consist of two DES keys, K_1 and K_2 , derived from the user passphrase. Conceptually, CFS file block encryption consists of encryption against a positionally defined stream cipher derived from K_1 , which is then encrypted with a codebook block cipher under K_2 , which is further encrypted with a second multi-use stream cipher derived from K_1 . Specifically,

$$E_p = DES^1(K_2, D_p \oplus DES^1(K_1, f(p \bmod m)) \oplus i) \\ \oplus DES^1(K_1, g(p \bmod m)) \quad (1)$$

The cipher is reversed in the obvious manner:

$$D_p = DES^{-1}(K_2, E_p \oplus DES^1(K_1, g(p \bmod m))) \\ \oplus DES^1(K_1, f(p \bmod m)) \oplus i \quad (2)$$

where:

E_p is the ciphertext block of a file at byte offset p .

D_p is the cleartext block of a file at byte offset p .

\oplus is the bitwise exclusive-or operation.

$DES^1(k, b)$

is the Data Encryption Standard block encryption function on cleartext b with key k .

$DES^{-1}(k, b)$

is the Data Encryption Standard block decryption function on ciphertext b with key k .

i is a bit representation of a unique file identifier derived from the Unix inode number and creation time of the encrypted file.

$f(n), g(n)$

are publically known functions that map an integer representation n into unique bit strings of the DES codebook size (64 bits).

- m is the length of the precomputed stored stream (presently 256K bytes).

Observe that the stream ciphers defined by $DES^1(K_1, f(p \bmod m))$ and $DES^1(K_1, g(p \bmod m))$ can be precomputed for each K_1 given $2m$ bytes of storage. The CFS daemon precomputes these streams when the cattach command is issued for a particular key. With the streams precomputed, each block encryption requires only one online DES operation (the codebook cipher based on K_2).

When decryption is performed on the card, the streams cannot be wholly precomputed in the card's small local memory. Instead, the card calculates $DES^1(K_1, f(p \bmod m))$ and $DES^1(K_1, g(p \bmod m))$ for each cipherblock sent to it. ($f(p \bmod m)$ and $g(p \bmod m)$ are sent to the card from the host computer as parameters with the cipher block.) Although this is computationally slower than the precomputed cipher, requiring three DES encryptions per block instead of one, bandwidth to the card interface (a serial link) remains the primary limitation on encryption speed.

4. Practical Applications

File system key escrow can support a variety of application domains. Ensuring organizational access to proprietary data was discussed and motivated above. Here, an employee has primary operational responsibility for data that belongs to an organization. Key escrow allows the organization to provide other individuals with emergency access capability in the primary employee's absence. Access by these "backup" individuals can be granted, controlled, audited and revoked easily, without compromising the organization's ability to maintain and control its own information.

Smartcard-based escrow also facilitates other backup access relationships. In the organizational scenario above, the primary key holder is "subordinate" to the escrow holder. Alternatively, a manager may be the primary key holder for sensitive-but-critical business data for which the keys are escrowed with an employee. The escrow key holder may not be authorized for routine access, but in the manager's absence may be required to perform "proxy" functions on the manager's behalf. Here, the smartcard system implements and enforces a common business delegation of authority practice.

Another scenario, which may become more important in the future, involves the protection of individual personal records. Consider, for example, a system in which medical records are encrypted under a key known only to the patient. Routine use of these records by a health practitioner requires the patient's active consent in supplying the key. In an emergency, however, access to the records may be required even when the patient is physically unable to supply the key. A key escrow smartcard, which might remain in the physical possession of the patient or be maintained with the records themselves, would enable such emergency access but still permit the patient to control (and revoke) the routine use of his or her private records. The proposed US national health care insurance system includes a smartcard-based identification token into which such a scheme could possibly be integrated.

4.1. Performance

The standard CFS system employs a software-based cryptographic engine that performs encryption on a modern workstation at between one and three Mbps[4]. Because CFS uses the standard file system cache, actual performance is much better, with a performance penalty of only 20-50% above the underlying file system under typical workloads. The escrow access system, on the other hand, performs all cryptographic operations on the smartcard, which communicates with the host workstation at serial link speeds (19,200 bps). After protocol and processing overhead, cryptographic bandwidth to the card is about 6,000 bps with the CFS cipher described in the previous section. Using the smartcard for decryption slows the cryptographic engine by almost three orders of magnitude. Cache performance hides this slightly, but the escrow access system is by no means transparent or fast enough for routine operational use.

In practice, the reduced performance is rarely an issue, since escrow access is not intended to support routine processing. (Write operations by the escrow agent are not even supported by our implementation). The normal mode of escrow operation involves copying out those files required for emergency access, such that the card is not subsequently required for their use.

These are not fundamental limitations. Faster smartcards are beginning to emerge in the market, along with faster interfaces with bandwidths that exceed the crypto-bandwidth of current software implementations. PCMCIA cards hold particular promise in this area.

4.2. Trust Model

Smartcard-based key escrow does not absolutely guarantee that the access policy will be enforced. There are risks associated with various parts of the system, each of which must be assessed in light of the application's security policy, threat model and available alternatives.

The system depends on the reverse engineering resistance of the escrow smartcard devices to control access by the escrow agents. Reverse engineering could reveal the keys stored on the card and permit the escrow agent to create duplicate cards without the knowledge of the key owner. Although the risk of reverse engineering is difficult to quantify as technology progresses, commercial smartcards are designed to resist this sort of attack. Recent trends in tamper-resistant packaging and chip fabrication technology suggest the emergence of future products with greatly reduced vulnerability to reverse engineering. In highly sensitive environments in which the integrity of the smartcard is not completely trusted, the card can be protected with augmented physical safeguards such as sealed envelopes and accountable paper audit trails.

By definition, the escrow agent has access to the escrowed data while in possession of the escrow card. The only built-in control on the escrow agent is access detection when the card is eventually audited. If the card is not returned by the agent, however, it is not possible to audit past access or prevent future access as long as the encrypted data remains available. The escrow key serves to limit unauthorized use of lost or stolen cards. If no single agent is trusted, possession of the card and the escrow key can be split among two or more agents. These risks are largely a function of the relationship between the escrow agent and the key owner. When appropriate, the owner can periodically audit the escrow card throughout the escrow period. Controls on access to the encrypted escrowed data can further ameliorate the risk of unauthorized access by the agent.

Any escrow system carries the risk of "cheating" by a key owner who encrypts data with keys other than those escrowed. This risk is present any time the key owner is able to supply his or her own cryptographic system. The check file in the smart-card system guards only against mistakes, not against deliberate deception. All escrow systems suffer from this limitation. In a centralized key distribution system, nothing prevents the use of "out of band" keys not obtained from the key center. In a system such as the government Escrowed Encryption Standard[8] (the "Clipper chip"), it is possible to suppress the

escrow exploitation field in the data stream or pre-encrypt with a secure non-escrowed cryptosystem. (The government system attempts to reduce this risk by supplying the escrowed devices in tamper-resistant modules, making it difficult to deploy the cipher without the escrow features.)

The risk of end-user escrow circumvention depends on the relationship between the key owner and the escrow agent. If escrow is perceived as a service for the mutual benefit of the key owner and agent, this risk is not an issue. If, on the other hand, this relationship is adversarial, there can be no completely reliable mechanism that prevents cheating.

5. Conclusions

Key escrow is not appropriate for all file encryption applications. Some data are simply too private; personal diaries, certain individual medical and financial records and other data for which there is no motivation for the data owner to allow third party access are poor candidates for escrow. Other data, such as day-to-day operational business records, have such high availability requirements to preclude any encryption at all. Escrow serves the "middle ground" for which security requirements suggest the need for cryptographic protection while availability requirements dictate the need for access.

Smartcard-based escrow overcomes the major shortcomings of software-based and manual escrow systems. Unlike manual systems, the escrowed keys can be reliably pre-audited to ensure their validity without compromising sensitive data. And unlike either system, once the card is returned, the owner is assured of whether the escrow process was used and that no further decryptions can occur. Escrowed decryption is completely under the control of the card; past possession of the card conveys no future privileges.

6. Acknowledgements

The author is indebted to Doug McIlroy for suggesting the encrypted file access problem. Jim Reeds' critical insights influenced the design of the CFS cipher. Eleanor Evans, Jack Lacy, Tom London and Adam Moskowitz made many helpful suggestions that improved this paper and the system it describes. We are particularly grateful to the anonymous referee who suggested medical records as an application area.

7. Availability

A research prototype of the base CFS system (implemented as a user-level NFS server) is available free upon request within the US and Canada. We regret that US Government-imposed export restrictions prevent us from making it available elsewhere. For information, `ftp dist/mab/cfs.announce` from `research.att.com` or send email to `cfs@research.att.com`. The smartcard software, including the escrow system described here, is not presently available.

8. References

- [1] Blaze, M., "A Cryptographic File System for Unix." *Proc. First ACM Conference on Computer and Communications Security*, Fairfax, VA, November 1993.
- [2] Denning, D. E., "Encryption and Law Enforcement." *Georgetown University, Computer Science Dept.*, Feb. 21, 1994, available by anonymous ftp from `cpsr.org`.
- [3] Diffie, W. and Hellman, M. E., "New Directions in Cryptography." *IEEE Trans. on Information Theory*, IT-11:644-654, November 1976.
- [4] Lacy, J., Mitchell, D. and Schell, W., "CryptoLib: Cryptography in Software." *Proc. Fourth USENIX Security Workshop*, October 1993.
- [5] Ioannidis, J. and Blaze, M., "Architecture and Implementation of Network-Layer Security Under Unix." *Proc. Fourth USENIX Security Workshop*, October 1993.
- [6] National Bureau of Standards, "Data Encryption Standard." *FIPS Publication #46*, NTIS, April 1977.
- [7] National Bureau of Standards, "Data Encryption Standard Modes of Operation." *FIPS Publication #81*, NTIS, December 1980.
- [8] National Institute for Standards and Technology, "Escrowed Encryption Standard." *FIPS Publication #185*, NTIS, February 1994.
- [9] Reiher, P., et. al., "Security Issues in the Truffles File System." *PSRG Workshop on Network and Distributed System Security*, 1993.
- [10] Rivest, R.L., Shamir, A. and Adleman, L., "A Method of Obtaining Digital Signatures and Public-Key Cryptosystems." *CACM*, February 1978.
- [11] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D. and Lyon, B., "Design and Implementation of the Sun Network File System." *Proc. USENIX*, Summer 1985.
- [12] Steiner, J., Neuman, C. and Schiller, J.I., "Kerberos: An Authentication Service for Open Network Systems." *Proc. USENIX*, Winter 1988.
- [13] Tygar, J.D. and Yee, B., "Strongbox: A System for Self Securing Programs." *CMU Computer Science: 25th Anniversary Commemorative*, Addison-Wesley, 1991.
- [14] Weiner, M.J., "Efficient DES Key Search." *Crypto '93*, (short presentation) August 1993.

Matt Blaze is a member of the Computing Systems Research Laboratory at AT&T Bell Laboratories, which he joined in 1992. His interests include secure systems, large-scale systems, network services, file systems, key management and privacy technology. Matt has a PhD in computer science from Princeton University and is a member of the adjunct faculty at Columbia University. You can reach him via email at `mab@research.att.com` or via postal mail at AT&T Bell Laboratories, 101 Crawfords Corner Rd, Room 4G-634, Holmdel, NJ 07733.

A Toolkit and Methods for Internet Firewalls

Marcus J. Ranum

Frederick M. Avolio

Trusted Information Systems, Inc.

Abstract

As the number of businesses and government agencies connecting to the Internet continues to increase, the demand for Internet firewalls — points of security guarding a private network from intrusion — has created a demand for reliable tools from which to build them. We present the TIS Internet Firewall Toolkit, which consists of software modules and configuration guidelines developed in the course of a broader ARPA-sponsored project. Components of the toolkit, while designed to work together, can be used in isolation or can be combined with other firewall components. The Firewall Toolkit software runs on UNIX® systems using TCP/IP with the Berkeley socket interface. We describe the Firewall Toolkit and the reasoning behind some of its design decisions, discuss some of the ways in which it may be configured, and conclude with some observations as to how it has served in practice.

Overview

Computer networks by their very nature are designed to allow the flow of information. Network technology is such that, today, you can sit at a workstation in Maryland, and have a process connected to a system in London, with files mounted from a system in California, and be able to do your work just as if all of the systems were in the same room as your computer. Impeding the free flow of data is contrary to the basic functionality of the network, but the free flow of information is contrary to the rules by which companies and governments need to conduct business. Proprietary information and sensitive data must be kept insulated from unauthorized access yet security must have a minimal impact on the overall useability of the network.

The purpose of an Internet firewall is to provide a point of defense and a controlled and audited access to services, both from within and without an organization's private network. This requires a mechanism for selectively permitting or blocking traffic between the Internet and the network being protected¹. Routers can control traffic at an IP level, by selectively permitting or denying traffic based on source/destination address or port. Hosts can control traffic at an application level, forcing traffic to move out of the protocol layer for more detailed examination. To implement a firewall that relies on routing and screening, one must permit at least a degree of direct IP-level traffic between the Internet and the protected network. Application level firewalls do not have this requirement, but are less flexible since they require development of specialized application forwarders known as "proxies." This design decision sets the general stance of the firewall, favoring either a higher degree of service or a higher degree of isolation. [1]

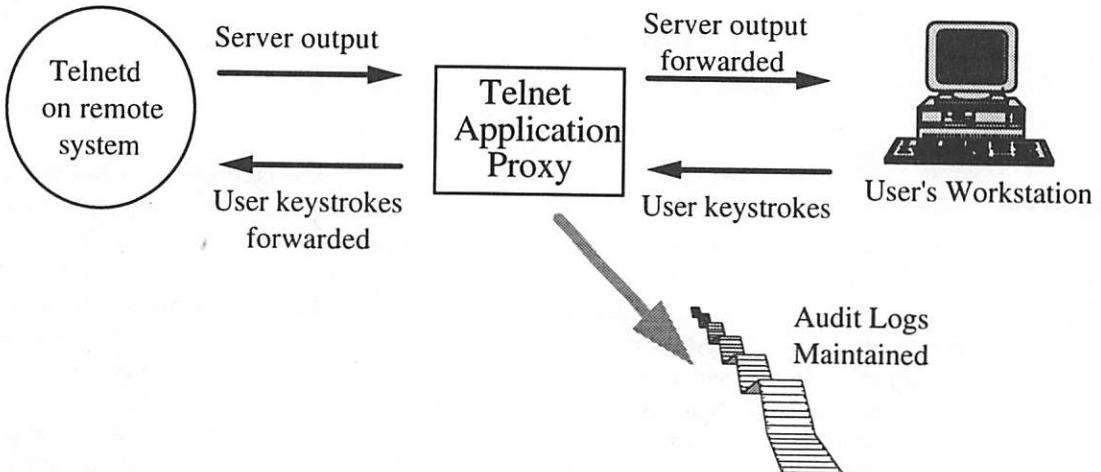
A proxy for a network protocol is an application that runs on a firewall host and connects specific service requests across the firewall, acting as a gateway. Figure 1 represents a minimal TELNET service proxy, in which the proxy forwards user's keystrokes to a remote system, and maintains audit records of connections. Proxies can give the illusion to the software on both sides of a direct point-to-point connection. Since many proxies interpret the protocol that they manage, additional access control and audit may be performed as desired. As an example, the FTP proxy can block FTP export of files while permitting import of files, representing a granularity of control that router-based firewalls cannot presently achieve. Router-based firewalls can provide higher throughput, since they operate at a

¹ Or, in general, between any two networks where one needs to be protected from the other.

protocol level, rather than an application level, but practical experience running firewalls on modern RISC processors shows that with a T-1 connection,

the bottleneck tends to remain the T-1 link rather than the firewall itself.

Figure 1: An Application Proxy



Proxies exist for a wide variety of services, such as X, FTP, TELNET, etc. Perhaps the most significant security benefit of employing proxies is that they provide a convenient opportunity to require authentication. For example, when connecting into a protected network from the Internet, one must typically first connect to the proxy, authenticate to it, and then complete a connection to a host within the protected network. The proxy protects the firewall host itself, by eliminating the need for the user to log into the firewall itself, and it protects the network by permitting only authenticated users to gain access from the outside. While hosts on the private network may still be rife with security holes, restricting the incoming traffic to authenticated users only is a good step in the right direction.

Other services, such as Internet (SMTP) mail and USENET news, act as store-and-forwarders already, and fit in with the proxy approach to firewalls. These service daemons sometimes run with system privileges and may contain bugs that an attacker can exploit. Many existing firewalls rely on approximate assessment of privileged systems software for their trustworthiness. This is sufficient if there are “well known working versions” of common programs such as the FTP server, *ftpd*. In some cases, however, the server can itself compromise security. A recent version of the WUArchive *ftpd*[2] contained a bug that permitted anyone on the Internet to gain super-user access to

systems on which it was running. In our design, we attempt to sidestep the issue by providing proxies that can run locked into a specific subdirectory by means of “*chroot*” — a UNIX system call that permanently restricts the working filesystem of a process. Proxies are also designed to run without special system privileges, to further reduce the chance that they might be able to damage the system. Ideally it should be impossible for an outside user to ever interact with a privileged process. Practically speaking, the Internet service master daemon *inetd*, which is responsible for starting other service daemons, needs to run with privileges, but outside users cannot interact directly with it. There is a possibility that the kernel may have trapdoors or hidden network services built into it, but it is impractical to attempt to obtain and examine kernel sources for such flaws. Instead, make every effort to remove unnecessary kernel services at system build time.

Design Philosophy

The TIS Firewall Toolkit (hereafter referred to as “the toolkit”) is designed to be informally verified for correctness as a whole or at a component level. Since the firewall consists of discrete components, each providing a single service, each may be examined separately from the rest of the system. Components of the toolkit are as simple as possible in their implementation, and are distributed in source code form to encourage peer review. This

appears to be a fairly novel approach for a network firewall, as many existing firewall systems rely on software that is "known to be good" or that is considered trustworthy because it has been used extensively for a long time.

One problem with the "known to be good" approach is that historically it hasn't been very reliable. Certain software components are frequently exploited in break-ins, no matter how carefully they are maintained. Problem programs are usually complex pieces of software, implemented in tens of thousands of lines of code, which require system privileges in order to operate. As a step towards addressing this, the firewall toolkit operates in accordance with the following general firewall design principles:

- Even if there is a bug in the implementation of a network service, it should not be able to compromise the system. Services that are misconfigured should not work at all, rather than opening holes.
- Hosts on the untrusted network should not be able to connect directly to network services that are running with privileges.
- Network services are implemented with a minimum of features and complexity. The source code is simple enough to be reviewed thoroughly and quickly.
- There should be reasonable and pragmatic means of testing that the system is correctly installed.

The toolkit is designed to be used with a host-based security policy, but its components can be used with router-based firewalls. In this paper, we will focus on the former. In a host-based firewall, the security of the host is crucial; once it is compromised the entire network is open to attack. Still, we believe that a host-based firewall is superior to other solutions because of the ease with which it can be maintained, configured, customized and audited. When the toolkit is used with router-based firewalls, it is assumed that the toolkit software is running on a secure host that is permitted some degree of access between the protected network and the Internet, by means of routers. This leaves the option of configuring the routers to provide additional avenues between the protected network and the Internet for whatever reason; such additional avenues are outside the scope of the toolkit and should be provided only after careful security analysis.

The toolkit may be used in conjunction with router-based screening as extra security. To

minimize risks, the services that are provided on the external machine, which we will refer to as a "bastion host", following the terminology proposed by Ranum[3], are sharply curtailed and each service is subjected to review. On the "standard" firewall configuration, the only services supported are SMTP, FTP, NNTP, and TELNET. Other proxies such as Digital Equipment Corporation's X Window System proxy [4] can be added to this architecture.

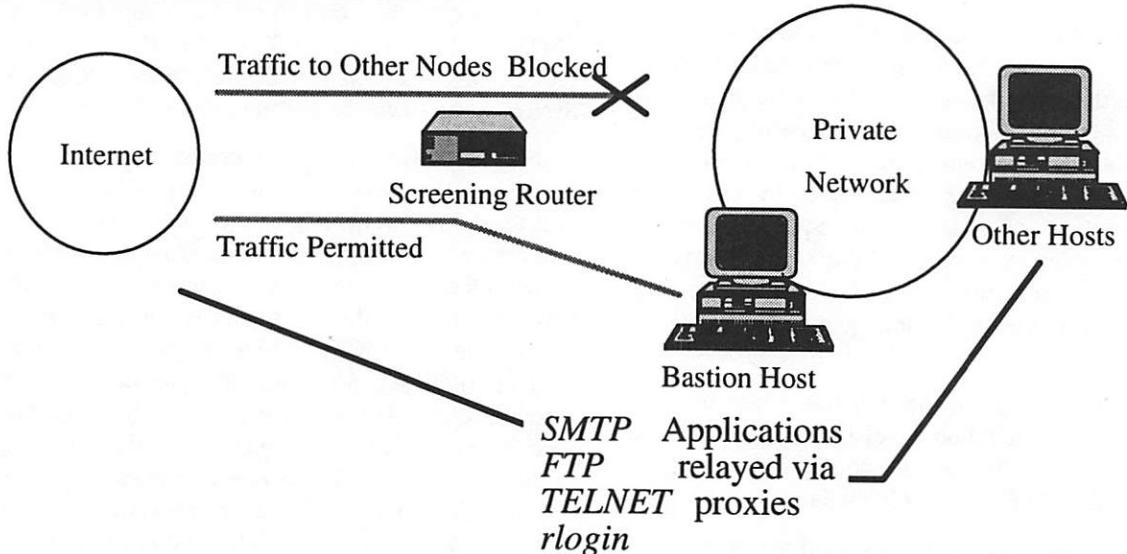
SMTP service is supported through a non-privileged front end that runs locked in a "safe directory" via chroot. FTP is supported via a proxy that runs without requiring special privileges. NNTP is supported via a "tunnel" server that permits traffic between a host on the inside and its news server on the outside. TELNET service is via a proxy that runs unprivileged. Since all other services on the system are disabled selectively, it is only these four services that must be analyzed for risk. By analyzing of the security of each service in isolation, we are able to gain a degree of trust in the system beyond merely being able to state "Well, we don't *think* there are any bugs." With all the services running unprivileged we can make a stronger statement, to wit, "The security of an individual service is irrelevant to the overall security, as the server is running in a captive mode."

Configuration and Components

Figure 2 represents the toolkit installed in an environment that combines routers and a firewall bastion host. The implementation of the security controls is shared (in this example) between the routers and the firewall: the routers are responsible for controlling network-level access, and the bastion host provides application-level control. A simpler firewall configuration would consist of a dual-homed gateway, in which a workstation with two network interfaces is connected to both networks, and has IP forwarding disabled. Dual homed gateways are less flexible than firewalls that combine routers and hosts, since the option to route services at a network level is generally not available.² On the other hand, with a dual-homed gateway, the administrator can have a higher degree of confidence that no network traffic will be able to somehow "leak" through a router, since routers are no longer an integral part of the security system.

² Some versions of UNIX support packet screening within the operating system.

Figure 2: A Screened Host Firewall



The toolkit is designed to build a host-based firewall, with security being enforced by a single bastion host. For ease of management, all the proxies and access control tools use a single configuration file with a regular syntax. We thought this was useful due to the generally complex configuration of various publicly available firewall tools, of which no two are configured in the same

way. The configuration rules are designed to provide both configuration and service and access permissions information, being read top-to-bottom and left-to-right. Hostnames or IP addresses including simple wildcards can be used in configuration rules, but IP addresses are preferred since DNS addresses are vulnerable to spoofing.

```
# Example ftp gateway rules:
# -----
ftp-gw: authserver      127.0.0.1 7777
ftp-gw: denial-msg       /usr/local/etc/ftp-deny.txt
ftp-gw: welcome-msg     /usr/local/etc/ftp-welcome.txt
ftp-gw: help-msg        /usr/local/etc/ftp-help.txt
ftp-gw: timeout          3600
ftp-gw: permit-hosts    192.33.112.100
ftp-gw: deny-hosts      128.52.46.*
ftp-gw: permit-hosts    192.33.112.* -log { retr stor } -auth { stor }
ftp-gw: permit-hosts    * -authall
```

The firewall toolkit functionality can be broken down into six areas: logging, electronic mail, the Domain Name Service, FTP, TELNET, and TCP access control.

Logging

Significant security events and audit records are logged to a protected host on the internal network via the syslog facility. The version of *syslogd* that the toolkit uses is based on the BSD

"net2" sources, with some modifications to support pattern-matching and program execution on matched patterns. Many systems administrators have batch processes set up on their systems to alert them of possible security problems by searching the system logs at regular intervals. By permitting the system manager to add regular expressions to the *syslogd* configuration, security-related log messages can be identified instantly. *Syslogd* contains further modifications that permit an arbitrary command to be invoked with any specified logging rule, so that,

for example, vitally important security log events can be delivered to the system manager's beeper or delivered by electronic mail. Adding command execution to *syslogd* implies that the *syslogd* configuration file must be protected against unauthorized modification.

Electronic Mail

Mailers are one of the favorite points of attack against UNIX systems. The Morris Internet worm exploited a well-known hole in the standard UNIX SMTP server, *sendmail*. Many systems running *sendmail*, including those with Internet firewalls, were penetrated by the worm. A few that had replaced *sendmail* with other SMTP servers were not. Since that time, a variety of other security holes have been identified in *sendmail* and fixed in more recent releases.

The problem with mailers is twofold: they are complex and perform file system activity, and they require privileges so that they can manipulate mailboxes or execute mail processing programs on the behalf of users. To help secure mail service, direct network access to *sendmail* is prevented. A simple program that implements a skeleton of the SMTP protocol is presented on the SMTP port on the mail server. This SMTP proxy, called *smap*, is small enough to be subjected to a code review for correctness (unlike *sendmail*) and simply accepts all incoming messages and writes them to disk in a spool area. Rather than running with permissions, the proxy runs with a restricted set of permissions and runs "chrooted" to the spool area. A second process is responsible for scanning the spool area and delivering the mail messages to the real *sendmail* for delivery — a mode of operation in which *sendmail* can operate with reduced permission. Many Internet firewalls run *sendmail* and rely on "trustworthy" versions of the software; running the mail software in a reduced-permissions mode is a more general solution to the problem, side-stepping the issue of whether or not a given version of *sendmail* contains bugs.

While *smap* answers all valid SMTP commands sent to it, it does not execute any of them except those directly involved with mail exchange: HELO, FROM, RCPT, DATA, and QUIT. Other commands, such as VRFY and EXPN return a polite error message. *Smap* preserves *sendmail*'s functionality, while preventing an arbitrary user on the network from communicating directly with it. Analyzing *sendmail*'s 20,000 lines of source code for

bugs is a sizable task when compared to analyzing *smap*'s 700 lines. *Smap* is not a panacea, however, as firewalls remain vulnerable to data-driven attacks in which messages may be mailed to hosts on the private network, possibly triggering security holes in internal mailers. Since many of these attacks have a distinctive signature, *smap* or the firewall's mailer can be configured to attempt to identify these letter-bombs, but the security administrator is forced into the unfortunate position of an arms-race in which a reactive role must be taken as new attacks are invented. To reduce the risk of attacks that exploit mailing through programs, the mailer on the firewall itself is configured so that program execution is disabled. Disabling program execution is often an unacceptable solution on a multi-user system, but since the firewall is not a general use host, we prefer to reduce the risk of someone being able to execute arbitrary commands from afar.

Domain Name Service (DNS)

The name service software available for UNIX implements an in-memory read-only database. As such, it cannot be used to gain unauthorized access to a system. Some past attacks on firewalls have used name service spoofing as a technique for impersonating trusted network hosts. In order to remove the threat of name service spoofing, the firewall does not rely on name service for any security related information. The name server software is necessary for high performance large-scale mail systems and is configured so that the only application that relies on name service for addressing is the electronic mail system. DNS names are also used in audit records, but are always presented along with host network addresses; mismatches are flagged as possible spoofing attempts.

FTP

The FTP application gateway is a single process that mediates FTP connections between two networks. Since it performs no disk access other than reading its configuration file and is a small and relatively uncomplicated program, it can be argued that it is not capable of compromising the security of the system. Just to be certain, the application gateway runs as a non-privileged user, after "chrooting" itself to a private directory on the system. To control FTP access, the application gateway reads a configuration file, containing a list of FTP commands that should be logged, and a description of what systems are allowed to engage in

FTP traffic. All traffic can be logged and summarized. Optionally, the gateway can permit FTP traffic from the Internet to the campus network for users who first authenticate themselves to the system.

TELNET

The TELNET application gateway is a small, simple application that mediates TELNET traffic. As with the FTP application gateway, the only file accessed is the configuration file that is read at start-up. Immediately after the configuration file is read, the TELNET application gateway is "chrooted" to a restricted directory, where it runs as a non-privileged process. The TELNET gateway's configuration file allows specification of which systems or networks can use it, and what systems or networks it will permit connection to. Initially, it will be configured to permit campus systems to use the gateway to connect to Internet systems, but not vice-versa. Optionally, the TELNET gateway can require authentication before permitting use. All connections and their durations are logged.

UDP-Based Services

Since we decided that no direct traffic would be permitted between an outside system and an inside system, and since UDP is connectionless and point-to-point (and so cannot be used through network proxies), UDP services are not allowed. Many UDP-based services such as NTP and DNS can be provided transparently through a firewall by configuring the servers to act as forwarders for queries originating within the protected network.

TCP Access and Use

On BSD-based UNIX systems, most network processes are started up by an initial connection to a general-purpose network listener *inetd*, which establishes a connection between the incoming request and the program to service the request. For example, an incoming request for the TELNET service is "heard" by the running network listener. The program, according to *inetd*'s configuration file and the entry for TELNET, is executed and connected to the incoming request.

Inetd, the Internet services daemon, performs no function other than to invoke specified processes to manage network services when a system attempts to connect to them. Some vendor implementations permit a systems administrator to specify the user-id that the service should be invoked

as, but there is no provision for limiting access based on the source of the request. A variety of implementations of "wrapper" processes are available on the Internet with varying functionality[5].

The toolkit uses a "wrapper" process called *netacl*, which provides support for all TCP-based services. (If only TCP-based services are supported, UDP services are disabled and are no longer a threat worth worrying about.) *Netacl* has no great advantages over other versions of TCP wrappers, other than its minimal size (240 lines of code, including a large copyright header and comments), its lack of support for UDP (purposely), and its sharing a common configuration mechanism with the other tools in the toolkit.

TCP Plug-Board Connection Server

Certain services such as Usenet news are often provided through a firewall. In such a situation, the administrator has the choice of either running the service on the firewall machine itself or installing a proxy server. Since running news on the firewall itself might expose the system to any bugs in the news software, it is safer to use a proxy to gateway the service onto a "safe" system on the campus network. *Plug-gw* is a general purpose proxy that "plugs" two services together transparently. Its primary use is for supporting Usenet news, but it can be employed as a general-purpose proxy if desired. *Plug-gw* is configurable, as are the other proxy servers. Since it only acts as a data pipe, it performs no local disk I/O and invokes no subshells or processes. Like the other proxy servers, it logs all connections.

Plug-boarding TCP connections through one's firewall should be undertaken with a degree of caution, since *plug-gw* uses no authentication other than the host address of the client, and does no examination of the traffic passing across it. In the case of NNTP, for example, a security flaw in the NNTP server on the internal host could still be exploited. The firewall will make it much harder for an attacker to gain access to the internal system to further exploit the hole; if the flawed NNTP server were running on the firewall bastion host itself, the entire firewall might be vulnerable. Alternate approaches, such as engineering the news server to run "chrooted" are potential areas for future research. From a standpoint of systems administration, we have found that news

administration is simplified by running it a readily accessible internal server.

User Authentication

The network authentication server *authsrv* provides a generic authentication service for toolkit proxies. Its use is optional, required only if the firewall FTP and TELNET proxies are configured to require authentication. *Authsrv* acts as a piece of "middleware" that integrates multiple forms of authentication, permitting an administrator to associate a preferred form of authentication with an individual user. This permits organizations that already provide users with authentication tokens to enable the same token for authenticating users to the firewall. A secondary goal of *authsrv* was to provide a simple programming interface for authentication service, since commercial authentication systems tend to have unique, nonstandard, interfaces. Several forms of challenge/response cards are supported, along with software-based one-time password systems, and plaintext passwords. Use of plaintext passwords over the internet is strongly discouraged, due to the threat of password sniffing attackers.

A simple administrative shell is included that permits the authentication database to be manipulated over a network, with optional support for encryption of authentication transactions. The *authsrv* database supports a basic form of group management; one or more users can be identified as the administrator of a group of users, and can add, delete, enable, or disable users within that group. *Authsrv* internally maintains information about the last time a user authenticated to the server and how many failed attempts have been made. It can automatically disable or time-lock accounts that have multiple failures. Extensive logs are maintained of all *authsrv* transactions. *Authsrv* is intended to run on a secured host, such as the bastion host itself, since its database must be protected from attack.

Testing Firewalls

Throughout the design of the toolkit, we tried to design each component so that it relied wherever possible on protections in the UNIX environment, rather than on elaborate code designed to check and deter threats. While the toolkit software doesn't include a test suite, it is designed to be easy to verify that each component operates as it is intended. As an example, the SMTP proxy smap runs "chrooted" to a subdirectory as an unprivileged process. It stands to reason that if the proxy performs

this operation properly, all files will be created in the proper directory, with the proper user permissions. If the administrator verifies that this is indeed the case, he can rely on the security of the operating system's support for "chroot" and user file permissions. By examining the assumptions of each service proxy, a degree of assurance that the firewall is well protected can be gained. This does not address the problem of possible bugs or protocol errors in the proxy implementations that might still permit a service to pass through the firewall. To attempt to address this, every effort is made to keep the implementation of the proxies, especially the parts that deal with access control, as simple as possible.

Firewall administration requires a seasoned UNIX systems manager. While the toolkit is fairly easy to install, it assumes an amount of expertise on the part of the administrator, since he must know how to interpret error conditions, configure the system, and disable potentially threatening services. While it is a temptation to make the toolkit software self-installing and self-configuring, doing so raises the possibility that someone might install it who lacks the basic skills necessary to know if they have in fact secured their network. Packaging the toolkit as a set of components that can be used freely has proven effective, since it fills a need on the part of those experienced system managers who would have had to design, write, debug, and test their own implementations if ours were not available.

Future Directions

In the future we will focus on the problem of adding newer interactive information retrieval services such as Gopher, WAIS and World Wide Web and broadcast services such as MBONE. Possible avenues for future research include integrating cryptography with the firewall software to permit firewall-to-firewall service and firewall-to-firewall authentication, possibly using kerberos protocols. Support for IP-on-demand services like PPP pose a problem for firewalls: is the dial-up user to be treated as an untrusted Internet host or as a part of the protected network? Adding support for authenticated and encrypted PPP service on the firewall itself is being examined.

Observations

In practice, we find that running servers without special system privileges increases our assurance that the firewall is secure. More

importantly, the methodology of turning off all services but a minimum, and then auditing each one on a case-by-case basis further increases confidence that the system is harder to break into. The basic design decisions in setting up a firewall (to route or not to route, to rely on the host or the router) remain unchanged, but the toolkit will work with either model.

Firewalls are a stop-gap measure that is needed because many services are developed that operate either with poor security or no security at all. Perhaps the most important lesson we can learn from firewalls is the need for strong session-level authentication in applications and well-designed application protocols.

Availability

The TIS Internet Firewall Toolkit is available in source form via anonymous FTP from [ftp.tis.com:/pub/firewall/toolkit/fwtk.tar.Z](ftp://ftp.tis.com:/pub/firewall/toolkit/fwtk.tar.Z). Information is available from the authors at fwall-support@tis.com. Send mail to fwall-users-request@tis.com to be added to the firewall toolkit user's mailing list. Future enhancements to the toolkit will be announced on *fwall-users* and other relevant mailing lists.

Acknowledgements

This work was done, in part, under a contract from the U. S. Department of Defense, Advanced Research Projects Agency (ARPA), number DABT 63-92-C-0020. [6]

References

- [1] Marcus J. Ranum, "Thinking About Firewalls," Proceedings of Second International Conference on Systems and Network Security and Management (SANS-II), April, 1993
- [2] Washington University Saint Louis, FTP server daemon. Available for FTP from wuarchive.wustl.edu
- [3] Marcus J. Ranum — "An Internet Firewall," Proceedings of First International Conference on Systems and Network Security and Management (SANS-I), Nov, 1992

[4] G. Winfield Treese and Alec Wolman, "X Through the Firewall, and Other Application Relays," Proceedings of USENIX Summer Conference, 1993. Also available as Cambridge Research Lab Technical Report 93/10, Digital Equipment Corporation, May 3, 1993.

[5] Wietse Venema, "TCP WRAPPER, network monitoring, access control, and booby traps," UNIX Security Symposium III Proceedings (Baltimore), September 1992

[6] Frederick M. Avolio and Marcus J. Ranum, "A Network Perimeter With Secure External Access," Internet Society Symposium on Network and Distributed Systems Security, February 1994.

William Cheswick, "The Design Of a Secure Internet Gateway," Proceedings of the 3rd USENIX Security Symposium, September 1992.

Stephen M. Bellovin and William Cheswick, "Firewalls and Internet Security: Repelling the Wily Hacker," Addison-Wesley, Spring 1994

Frederick M. Avolio is a principal analyst with Trusted Information Systems, Incorporated, and active in network security consulting and product development. He has lectured on the subject of Internet gateways and firewalls and electronic mail configuration and has performed consulting services in these areas, both for government and in the private sector. He has worked in the UNIX and TCP/IP communities since 1979.

Mr. Avolio has an undergraduate degree in Computer Science from the University of Dayton and a Master of Science from Indiana University.

Marcus Ranum is a senior scientist at Trusted Information Systems. He is the chief architect of the firewall toolkit and spends most of his time on Internet security issues.

UNIX is a registered trademark of X/Open Company, Ltd.

SNP: An Interface for Secure Network Programming*

Thomas Y.C. Woo, Raghuram Bindignavle, Shaowen Su and Simon S. Lam

Department of Computer Sciences

The University of Texas at Austin

Austin, Texas 78712-1188

Abstract

SNP provides a high-level abstraction for secure end-to-end network communications. It supports both stream and datagram semantics with security guarantees (e.g., data origin authenticity, data integrity and data confidentiality). It is designed to resemble the Berkeley sockets interface so that security can be easily retrofitted into existing socket programs with only minor modifications. SNP is built on top of GSS-API, thus making it relatively portable across different authentication mechanisms conforming to GSS-API. SNP hides the details of GSS-API (e.g., credentials and contexts management), the communication sublayer as well as the cryptographic sublayer from the application programmers. It also encapsulates security sensitive information, thus preventing accidental or intentional disclosure by an application program.

1 Introduction

The explosive growth of network connectivity has significantly aggravated the problem of security. Most existing network programming paradigms adopt a trust-based approach to security (e.g., trusting network packets, trusting hosts). This is no longer adequate, especially for malicious attacks. Indeed, with easy access to networks and availability of sophisticated network tools, the effort to mount attacks such as spoofing network packets or sniffing illicit information from network traffic is substantially reduced. To effectively counter these attacks, a coherent security infrastructure is needed. An important element of such a infrastructure is a convenient abstraction for secure application network programming.

In recent years, distributed systems security has received a great deal of attention. For example, a number of authentication systems (e.g., Kerberos from MIT [15], SPX from DEC [17] and KryptoKnight from IBM [9]) have been designed and implemented. Although these systems

do provide an adequate solution for typical network security concerns, they suffer a major common drawback, namely, it is difficult to integrate them into an application. More specifically, they do not export a clean and easy-to-use interface that can be readily used in implementing a distributed service. For example, it often takes a considerable amount of effort to "kerberize" an existing distributed service. Besides, the interface provided is not portable, making the switch from one authentication system to another a non-trivial task.

The recently published Internet draft standard *Generic Security Service Application Program Interface* (GSS-API) [8] alleviates the problem somewhat. In fact, both SPX and KryptoKnight¹ have already implemented a small subset of GSS-API.² However, the GSS-API interface is still too low-level to be practical for typical network application programming. Its proper use requires intimate understanding of the underlying GSS-API concepts, which can cause significant distraction from the main task of a program. It is valid to say that GSS-API is more suited for use in system software than in regular application programming. Indeed, it is intended that a typical caller of GSS-API be a communication protocol, e.g., telnet, ftp [8, p. 2].

We believe that what is needed is an abstraction for secure network programming that can hide most of the details of GSS-API while retaining the same ease of use as most existing abstractions for network programming. As an analogy, the raw interface to a protocol (e.g., `tcp_input()`/`tcp_output()` for TCP) is often difficult to use, whereas programming using a higher-level abstraction (e.g., sockets, TLI) is significantly easier.³

In this paper, we discuss the design and implementation of SNP (*Secure Network Programming*), a high-level

¹KryptoKnight is not public-domain. The use of GSS-API is mentioned in [9]. But it is not clear to what extent the interface has been implemented.

²A recent article in *comp.protocols.kerberos* states that implementations of GSS-API for Kerberos will also be available.

³In fact, Berkeley sockets have often been touted as a major contributing factor to the popularity of TCP/IP. Although Berkeley sockets can support a variety of protocols, it was designed mainly with TCP/IP in mind.

*Research supported in part by NSA INFOSEC University Research Program under contract no. MDA 904-91-C7046 and MDA 904-93-C4089, and in part by National Science Foundation grant no. NCR-9004464.

abstraction for secure network programming that we have developed. SNP is like sockets or TLI in that it is an interface that provides applications access to network communications. However, it differs from sockets or TLI in many significant ways:

- SNP provides *secure* network communication. For example, it provides data origin authenticity, data integrity and data confidentiality services on top of the usual *stream* and *datagram* services provided by sockets or TLI. The precise services provided by SNP are detailed in Section 4.
- SNP provides an end-to-end communication abstraction at the application level, whereas sockets and TLI are transport level abstractions.⁴ More specifically, a socket represents a transport level endpoint (e.g., a TCP port), while an SNP endpoint represents an application layer entity (e.g., a server). This distinction is important and is further explained in Section 3.

SNP is implemented on top of GSS-API. It is currently in the form of a library. It adopts the same basic design as sockets (though several new calls have been added), which allows easy transitions from socket-based programs.

The balance of this paper is organized as follows. In the next section, we present an overview of SNP. This provides a quick introduction to SNP before delving into details in later sections. In Section 3, we elaborate on a list of design requirements and decisions we have made in the design of SNP. In Section 4, we provide a high-level description of the services offered by SNP. In Section 5, we give a specification of the SNP interface. In Section 6, we discuss various considerations that arise in implementing SNP. In Section 7, we provide some figures on the performance of our implementation. In Section 8, we compare SNP to some related systems. In Section 9, we discuss the lessons learned and directions for future work.

2 Overview of SNP

2.1 A Quick First Look

To give a quick introduction of what SNP is, we begin by looking at actual SNP code fragments. Figures 1 and 2 show respectively the typical client and server SNP code.

As can be easily seen, the SNP interface closely resembles that of sockets. This resemblance is not a coincidence. Rather, it was a design decision (see Section 3 for the rationale). In fact, most of the calls even retain their familiar

⁴This is not strictly true as sockets also provide access to protocols in other layers in the communication hierarchy, cf., *raw sockets* etc. However, sockets and TLI are typically considered to be transport layer interfaces.

semantics from their socket counterparts, though their implementations are quite different. In the following, we will focus only on the calls that are new in SNP.

There are two main new calls, namely `snp()` and `snp_attach()`. `snp()` replaces the `socket()` call in the socket interface. It is similar in functionality to `socket()` in that it creates a communication endpoint. It differs from `socket()`, however, in that an SNP endpoint corresponds to an application layer entity rather than a transport layer entity.

In addition to `SOCK_STREAM` and `SOCK_DGRAM`, `snp()` supports two new kinds of communication semantics: `SNP_STREAM` and `SNP_DGRAM`. Both extend the semantics of their respective socket counterparts by adding security guarantees. Specifically, an `SNP_STREAM` connection is authenticated. That is, a connection would be made only if it is accepted by the intended peer (specified by the initiator); and conversely, the identity of the initiator can be uniquely determined by the intended peer once a connection is made. Additional security services (e.g., data integrity, data confidentiality) can be activated on an `SNP_STREAM` connection by setting the appropriate options using `snp_setopt()` (see Section 4). Essentially, an `SNP_STREAM` connection can be understood as a connection that supports the semantics of a `SOCK_STREAM` connection even in an environment with intruders.⁵ The case for `SNP_DGRAM` is similar.

`snp_attach()` is a completely new call; it does not have a socket counterpart. The main function of this call is to attach an *identity* to an SNP endpoint. The attached identity is the one that would be authenticated to a peer. An identity is not just a name, it is a *supported* claim of a particular name.⁶ In other words, an identity can be unambiguously verified to another party. In terms of implementation, an identity consists of a name together with a set of *credentials* that corroborate the authenticity of the name. Typically, the operation of `snp_attach()` involves collecting the appropriate credentials (locally and/or remotely) for supporting the specified name. It should be noted that the identity attached to an SNP endpoint needs not be the identity of the caller *per se*. For example, a delegate may want to attach its identity as a delegate rather than its own identity. Operationally, a caller is allowed to attach any identity to an endpoint as long as it is able to gather the required credentials to support that identity.

⁵Assuming proper options are set corresponding to the types of threats anticipated.

⁶A better term for this is *principal*. But we intend to keep it informal in this paper and refrain from introducing too many terms.

```

#include <snp.h>
...
if ((snp_ep =.snp(AF_INET, SNP_STREAM, SNP_PROTO_DEFAULT)) < 0) {
    snp_perror("snp() error :");
    exit(1);
}
/* Initialize local and peer addr structs - just as in sockets */
...
/* Initialize local & peer name structs as shown below */
local_name.np.np_val      = (char *) malloc(sizeof(client_name));
local_name.np.np_len       = strlen(client_name);
strcpy(local_name.np.np_val, client_name);
peer_name.np.np_val        = (char *) malloc(sizeof(server_name));
peer_name.np.np_len        = strlen(server_name);
strcpy(peer_name.np.np_val, server_name);

if (snp_attach(snp_ep, &local_name, &peer_name) < 0) {
    snp_perror("snp_attach() error :");
    exit(1);
}
if (snp_connect(snp_ep, sizeof(struct sockaddr_in),
                (struct sockaddr *) (&peer_addr) ) < 0) {
    snp_perror("snp_connect() error :");
    exit(1);
}
if ((numbytes = snp_write(snp_ep, buf, buf_size)) < 0) {
    snp_perror("snp_write() error :");
    exit(1);
}
...
if (snp_close(snp_ep) < 0) {
    snp_perror("snp_close() error :");
    exit(1);
}
...

```

Figure 1: Sample SNP Client Program Fragment

```

#include <snp.h>
...
if ((snp_ep =.snp(AF_INET, SNP_STREAM, SNP_PROTO_DEFAULT)) < 0) {
    snp_perror("snp() error :");
    exit(1);
}
/* Initialize local and peer addr structs - just as in sockets */
...
/* Initialize local name structs as shown below */
local_name.np.np_val      = (char *) malloc(sizeof(server_name));
local_name.np.np_len       = strlen(server_name);
strcpy(local_name.np.np_val, server_name);
if (snp_attach(snp_ep, &local_name, &peer_name) < 0) {
    snp_perror("snp_attach() error :");
    exit(1);
}
if (snp_bind(snp_ep, &server_addr, sizeof(server_addr)) < 0) {
    snp_perror("snp_bind() error :");
    exit(1);
}
if (snp_listen(snp_ep, 5) < 0) {
    snp_perror("snp_listen() error :");
    exit(1);
}
if ((new_snp_ep = snp_accept(snp_ep, (struct sockaddr *) &peer_addr,
                             &addr_len)) < 0) {
    snp_perror("snp_accept() error :");
    exit(1);
}
if (snp_getpeerid (snp_ep, &client_name) < 0) {
    snp_perror("snp_getpeerid() error :");
    exit(1);
}
if ((numbytes = snp_read(new_snp_ep, buf, buf_size)) < 0) {
    snp_perror("snp_read() error :");
    exit(1);
}
...
if (snp_close(new_snp_ep) < 0) {
    snp_perror("snp_close() error :");
    exit(1);
}
...
if (snp_close(snp_ep) < 0) {
    snp_perror("snp_close() error :");
    exit(1);
}
...

```

Figure 2: Sample SNP Sequential Server Program Fragment

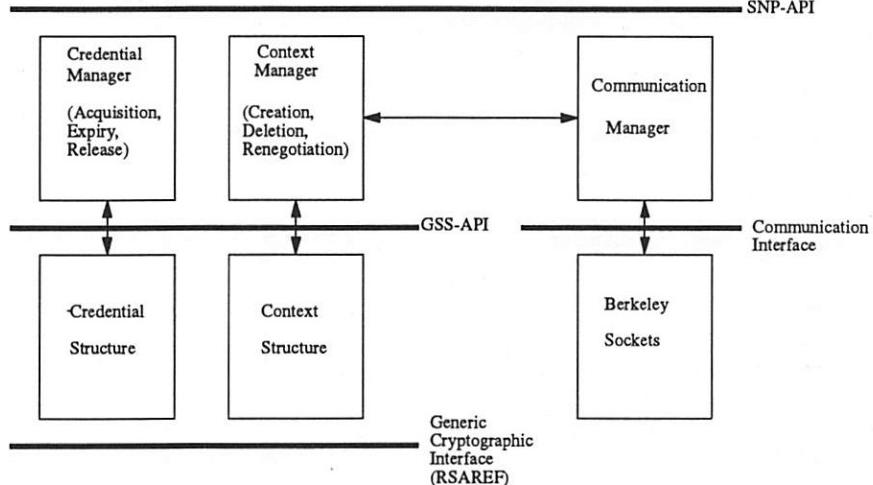


Figure 3: Organization of SNP

2.2 Components of SNP

SNP is designed and implemented in a modular fashion. Each major functionality of SNP is encapsulated in a separate layer that exports a well-defined interface. Figure 3 shows the conceptual layering and major components of SNP.

SNP-API defines the upper (external) interface available to an application. Internally, SNP makes use of two lower interfaces: GSS-API and a (insecure) network communication API. GSS-API encapsulates the details of the particular authentication protocol used, thus enhancing the independence of the SNP layer from the underlying authentication mechanism. Similarly, the communication API isolates the details of network communication from the SNP layer. We have chosen sockets as the communication API, mainly due to its wide availability.

GSS-API in turns makes use of a lower generic cryptographic interface. This interface provides access to all cryptographic functions and is generic in the sense that it can support any (symmetric or asymmetric) cryptosystem. This provides “cryptosystem independence” and facilitates easy substitution when new (implementations of) cryptosystems are available. Further discussion of our GSS-API implementation and the underlying authentication protocol is beyond the scope of this paper; interested readers can consult [23, 20] for more details.

The main function of the SNP layer is *context* and *credential management*. It initiates the acquisition of credentials, monitors the status of contexts and credentials, and initiates renegotiation (of contexts) and/or reacquisition (of credentials), if necessary. It should be noted, however, that the actual storage of contexts and credentials is internal to GSS-API.

2.3 SNP in Context

SNP is part of a larger project of ours that concerns the design and implementation of an *authentication framework* for distributed systems [21]. The framework addresses a range of authentication needs that includes bootstrapping, user logins and peer communications. SNP is designed as an interface for accessing the peer authentication protocol in our framework.

Because of its modular design, detailed understanding of the other components in the framework is not required in order to use or understand SNP. Indeed, SNP is relatively independent of the original framework it was designed for, and should be easily portable for use in other authentication frameworks (see Section 3). Therefore, we will only briefly describe the other components in our framework, to the extent they are required for the understanding of SNP.

At present, our framework has three protocols: a *secure bootstrap protocol* that creates a *bootstrap certificate* upon successful bootstrapping; a *user-host mutual authentication protocol* that creates a *login certificate* when a user successfully logs in; and a *peer-peer mutual authentication protocol* that is the basis for SNP. The login certificate is retrieved when a user *attaches* its identity to an SNP endpoint. This certificate is stored in an SNP (GSS-API) credential structure and is used to authenticate the user’s identity to its peer.

The peer-peer authentication protocol in our framework assumes the use of a commonly trusted *authentication server* (AS). Apart from its authentication duty, AS is also responsible for generating the session key used in an authentication exchange. Thus, in order for SNP to function correctly, AS needs to be properly set up. For example, it must be properly secured and be given a correct database. The discussion of the associated administrative issues is beyond the scope of this paper.

Lastly, a *name service* is required for translating application layer entities to their transport layer addresses. This name service, however, need not be trusted, as SNP performs the proper authentication during connection establishment.

3 Design Requirements and Decisions

In designing SNP, we first set out a number of requirements. Based on these requirements, we made several key design decisions. We briefly discuss the rationale for these requirements and decisions below:

- SNP should provide end-to-end communication at the application layer rather than the transport layer. Although the transport layer is the first end-to-end layer, we believe the concept of identity is only meaningful at or above the session layer. For example, in Unix and TCP/IP, ports are ephemeral and the association of ports with processes is dynamic.⁷ We believe it is more appropriate to base our semantics on application level entities than to assume a secure mapping between ports and processes.
- SNP should be independent of any particular authentication protocol or framework. This allows SNP to be portable across different authentication systems. We achieve such independence by using GSS-API to encapsulate the details of the underlying protocol, and sockets as the communication interface.
- It should be easy to convert existing network application programs to use SNP. To achieve this, we designed SNP to retain as much as possible the general structure of a socket program. Hence: (1) only a minimal number of new concepts needs to be learned in order to acquaint oneself with SNP; (2) only minor (mostly syntactic) modifications need to be done to convert a socket program to an SNP program, thus significantly facilitating retrofitting.

We could have emulated the TLI interface instead. But we believe that sockets and TLI are sufficiently similar to each other that little extra effort is required to convert TLI programs into SNP programs. Besides, there are far more existing socket programs than TLI programs, though TLI is quickly gaining popularity.

- SNP should work in a heterogeneous environment. This entails careful considerations of message encoding and processing. We have chosen XDR [16]

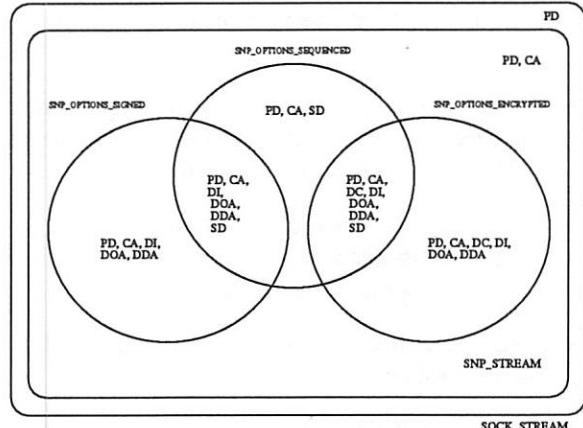


Figure 4: Services Provided

for this purpose, mainly for its simplicity. ASN.1 [1] is used in other authentication systems (e.g., Kerberos, SPX); we find it to be overly complicated and not suitable as a prototyping tool. From our experience, XDR has been adequate, though not as flexible as we would like.

- SNP should be independent of particular cryptosystems. We achieve this by encapsulating all cryptographic functions using a generic cryptographic interface. In our current implementation, we use the *de facto* standard cryptosystem trio, i.e., DES [10] for symmetric encryption, RSA [13] for asymmetric encryption and MD5 [12] for message digest.

4 Services Provided

Security is only well-defined with respect to a threat model. In this paper, we assume the standard threat model. That is, a saboteur can read, insert, delete and modify any network traffic. It should be noted that a saboteur is not necessarily a totally external intruder, s/he can also be a legitimate user. Thus, s/he can use information available to a legitimate user in mounting an attack.

We stress that our model does not include *denial of service* and *traffic analysis* threats. It is always possible for a saboteur to corrupt all packets passing through. Even an infinitely persistent sender cannot overcome such corruption if the saboteur is equally persistent. Indeed, most network programming abstractions guarantee only *safety* but not *progress*.

In the following, we present in high-level terms the services provided by SNP. We first define below the typical types of services offered by a secure communication connection:

- *Persistent Delivery* (PD)—A sender will persistently try to retransmit data if it has not been received yet.

⁷Reserved ports are a matter of convention only, there is no permanent binding.

Thus, PD implicitly assumes the use of acknowledgments.

- *Best Effort Delivery* (BED) — Data sent may or may not arrive at the receiver. Each of the intermediate nodes can either forward or drop the data.
- *Sequenced Delivery* (SD) — If data arrives at a receiver, it must appear in the same order it was sent. That is, no reordering or duplication is allowed.
- *Data Confidentiality* (DC) — Data is only legible to the intended receiver.
- *Data Integrity* (DI) — Data, if accepted by a receiver, must bear the same content as that sent.
- *Data Origin Authenticity* (DOA) — Data, if accepted by a receiver, must have come from a known specific sender.
- *Data Destination Authenticity* (DDA) — When data arrives, a receiver can unambiguously determine that it is the intended receiver.
- *Connection Authenticity* (CA) — A connection, if made, must be between the intended peers.

SNP can provide different combinations of the above services. The precise combinations provided is summarized in Figure 4. Each of the two boxes is labeled by a constant denoting the communication semantics, while each of the circles is labeled by an SNP option constant. The combination of services provided under a particular communication semantics and set of options is labeled in the intersection of the corresponding regions. For example, under `SNP_STREAM`, if both `SNP_OPTIONS_SIGNED` and `SNP_OPTIONS_SEQUENCED` are set, the services provided are PD, CA, DI, DOA, DDA and SD. We note that `SNP_OPTIONS_SIGNED` and `SNP_OPTIONS_ENCRYPTED` cannot both be set at the same time. The case for `SNP_DGRAM` is similar, and is omitted.

5 The SNP Interface

As with the socket interface, SNP-API functions can be divided into five classes: initialization, connection establishment, data transfer, connection release, and utility. We describe the functions in each class below. A complete list of all functions is given in Figure 5. Parameter names appearing in the following subsections refer to those shown there.

Most functions have semantics similar to their socket counterpart. (In fact, they are given the same names modulo the prefix “`snp_`.”) We have not emulated all the data

transfer functions of sockets (e.g., `sendmsg`, `recvmsg`) due to their intricate semantics. Nonblocking I/O is supported, but asynchronous I/O (i.e., interrupt driven) is not.

We also note that most functions below (notable exceptions being the data transfer functions) return 0 on success and -1 on failure. In addition, a global variable `snp_errno` will contain the appropriate error number on failure.

5.1 Initialization

Functions in this class are used for creating and initializing an SNP endpoint. They include `snp()`, `snp_bind()`, `snp_listen()` and `snp_attach()`.

5.1.1 `snp()`

`snp()` creates an endpoint of communication. Its parameters have the same types as `socket()` and have similar semantics. Currently, the only supported value for `family` is AF_INET, corresponding to the internet address family. The possible values of `type` are shown in the following table:

<code>SNP_STREAM</code>	Secure Stream
<code>SNP_DGRAM</code>	Secure Datagram
<code>SOCK_STREAM</code>	Normal (Insecure) Stream
<code>SOCK_DGRAM</code>	Normal (Insecure) Datagram

For `protocol`, the currently supported values are as follows:

<code>SNP_PROTO_DEFAULT</code>	Default Authentication Protocol
<code>SNP_PROTO_PUSH_MODEL</code>	Push Model Authentication Protocol
<code>SNP_PROTO_REVERSE</code>	Reverse Authentication Protocol
<code>IPPROTO_TCP</code>	Normal TCP
<code>IPPROTO_UDP</code>	Normal UDP

A combination of `SNP_STREAM` and any one of the first three protocol values results in a secure equivalent of TCP. Similarly, `SNP_DGRAM` in combination with one of the first three protocol constants provides a secure UDP protocol. The first three protocol constants can be used only when the `family` argument value has been set to either `SNP_STREAM` or `SNP_DGRAM`. The use of either `IPPROTO_UDP` or `IPPROTO_TCP` results in the normal (i.e., insecure) UDP or TCP protocols, respectively. These are equivalent to the semantics provided by the socket interface.

`snp()` returns an SNP handle, of type `int`. The handle is an index into an internal table of SNP structures maintained by SNP. Thus, unlike `socket()`, an SNP handle is not a file descriptor. Hence, some of the standard functions that apply to a socket descriptor will not apply to an SNP handle.

The `snp_ep` parameter in each of the other functions in Figure 5 refer to an SNP handle obtained from a call to `snp()`.

```

Initialization Calls
int.snp_bind      ( int family, int type, int protocol );
int.snp_listen    ( int.snp_ep, struct.sockaddr *local_addr, int.addr_len );
int.snp_attach    ( int.snp_ep, struct.name_s *local_name, struct.name_s *peer_name );

Connection Establishment Calls
int.snp_connect   ( int.snp_ep, struct.sockaddr *peer_addr, int.peer_addr_len );
int.snp_accept    ( int.snp_ep, struct.sockaddr *peer_addr, int.peer_addr_len );

Data Transfer Calls
int.snp_write     ( int.snp_ep, char *buf, int.nbytes );
int.snp_read      ( int.snp_ep, char *buf, int.nbytes );
int.snp_send      ( int.snp_ep, char *buf, int.nbytes, int.flags );
int.snp_recv      ( int.snp_ep, char *buf, int.nbytes, int.flags );
int.snp_sendto    ( int.snp_ep, char *buf, int.nbytes, int.flags,
                    struct.sockaddr *to, int.tolen );
int.snp_recvfrom  ( int.snp_ep, char *buf, int.nbytes, int.flags,
                    struct.sockaddr *from, int.*fromlen );

Connection Release Calls
int.snp_close     ( int.snp_ep );
int.snp_shutdown  ( int.snp_ep, int.how );

Utility Calls
int.snp_setopt    ( int.snp_ep, int.level, int.optname, char *optval, int.optlen );
int.snp_getpeerid ( int.snp_ep, struct.name_s *peer_name );

```

Figure 5: SNP Interface Specification

5.1.2 `snp_bind()`

After creation, an address may be bound to an SNP endpoint using `snp_bind()`. The `local_addr` and `addr_len` are of the same types as in the `bind()` function. They specify the address to be bound.

5.1.3 `snp_attach()`

`snp_attach()` is used for specifying the identity a caller wishes to be authenticated as to its peer and the name of the intended peer. The name structure `name_s` is of the following form: (This structure is automatically generated by `rpcgen` from a XDR structure.)

```

struct.name_s {
    struct {
        u_int np_len; /* Length of the name */
        char *np_val; /* The actual name */
    } np;
};

```

If invoked by a server, `peer_name` may be set to `NULL`, in which case connection from any client would be accepted. Once a connection is established, the identity of the client can be discovered by calling `snp_getpeerid()` (see below). `snp_attach()` must be invoked before connection establishment, if secure communication is desired.

5.1.4 `snp_listen()`

The function allows its caller to specify the maximum allowed backlog of connection requests. It has identical

semantics as `listen()`, except it takes an SNP handle. Typically, a caller of `snp_listen()` is a server. This function can only be used on an SNP_STREAM or SOCK_STREAM connection.

5.2 Connection Establishment

The second class of functions consists of `snp_connect()` and `snp_accept()`; they are mostly used for stream connections.

5.2.1 `snp_connect()`

For an SNP_STREAM endpoint, this function results in the establishment of a connection with a peer if a corresponding `snp_accept()` is performed by the peer. A successful connection also indicates a successful authentication exchange using the underlying authentication protocol.

In the case of SNP_DGRAM, `snp_connect()` only saves the supplied peer address in an internal SNP structure. This address would be assumed to be the destination address in all subsequent data transfer unless an explicit address is given. No authentication is performed at the time of the call; instead, it is performed at the time of the first data transfer call.

5.2.2 `snp_accept()`

`snp_accept()` can be used only on an `SNP_STREAM` or `SOCK_STREAM` endpoint. It accepts connection requests and completes them if the authenticated peer identity matches the one specified by a previous `snp_attach()`.⁸ Successful completion also implies that the peer identity has been authenticated, and can be discovered using `snp_getpeerid()`. Furthermore, it implies the establishment of a pair of security contexts (one at each peer) and the distribution of a session key.

The return value is a new SNP handle which can be used for further communication with the peer. Further connection requests can continue to come in on the original SNP endpoint. If `peer_addr` and `peer_addr_len` are non-NULL, they will be filled in appropriately.

5.3 Data Transfer

All of the following data transfer functions return the number of bytes actually sent or received on success and -1 on failure.

5.3.1 `snp_sendto()`

`snp_sendto()` sends `nbytes` of data pointed to by `buf` to the peer address specified by the `to` parameter. This function may be used on both stream and datagram endpoints. In case of a datagram endpoint, both `to` and `to_len` must be specified. The data will be sent encrypted or signed if the appropriate SNP options have been set (see `snp_setopt()` below). The possible values and semantics of `flags` are the same as those in `sendto()`.

5.3.2 `snp_recvfrom()`

`snp_recvfrom()` attempts to receive `nbytes` of data and stores them in a buffer pointed to by `buf`. The address and address length of the peer are filled into `from` and `from_len` respectively, if both of them are non-NULL. `flags` has the same semantics as in the `recvfrom()`. The incoming data is decrypted or verified, depending upon the SNP options specified.

5.3.3 `snp_read()`, `snp_write()`, `snp_send()` and `snp_recv()`

These calls can only be used on stream endpoints. Their semantics are essentially similar to their socket counterparts. `snp_send()` and `snp_recv()` provides additional features (e.g., such as expedited data) that are not available with `snp_write()` and `snp_read()`. The nature of data sent or received depends on the current SNP options.

⁸If the peer name specified is NULL, connections from any client is accepted.

5.4 Connection Release

5.4.1 `snp_shutdown()` and `snp_close()`

These functions have similar semantics as their socket counterparts, except they perform the release only after they have verified that the release request did originate from the correct peer.

5.5 Utility Routines

These functions are used for manipulating or retrieving the characteristics of an SNP endpoint.

5.5.1 `snp_setopt()`

`snp_setopt()` is used to set options available for a regular socket as well as those specific to SNP. A new constant, `SNP`, has been introduced for the `level` parameter. The options available at the SNP level are:

<code>SNP_OPTIONS_DEFAULT</code>	Reset all option settings to default
<code>SNP_OPTIONS_ENCRYPTED</code>	Encrypt all subsequent data
<code>SNP_OPTIONS_SIGNED</code>	Sign all subsequent data
<code>SNP_OPTIONS_SEQUENCED</code>	Enforce sequencing on data
<code>SNP_OPTIONS_NOTIFY</code>	Notify caller on context expiry — do not reinitiate authentication
<code>SNP_OPTIONS_CONTEXT_TIME</code>	Set context expiration time

Setting `SNP_OPTIONS_DEFAULT` results in resetting all options to their default settings; that is, no encryption, no signing and no sequencing.

Setting `SNP_OPTIONS_ENCRYPTED` causes subsequent outgoing data to be encrypted. Setting `SNP_OPTIONS_SIGNED` causes subsequent outgoing data to be signed. The key to be used for encryption and signing is the session key maintained in the current security context. Options `SNP_OPTIONS_ENCRYPTED` and `SNP_OPTIONS_SIGNED` cannot be set at the same time. To enforce sequencing of data, option `SNP_OPTIONS_SEQUENCED` should be set. This may be used in conjunction with either `SNP_OPTIONS_ENCRYPTED` or `SNP_OPTIONS_SIGNED`.

When the current security context expires, the SNP layer automatically renegotiates a new context. This can be disabled by setting `SNP_OPTIONS_NOTIFY`; in which case, the SNP user will be notified of context expiry when it performs an SNP call. The duration of a context can be set using the `SNP_OPTIONS_CONTEXT_TIME` option.

Note that the first five options are toggle flags, while the last one requires the context duration to be specified in `optval`.

5.5.2 `snp_perror()` and `snp_getpeerid()`

`snp_perror()` performs the same function as the standard `perror()` function, except that it accounts for

Connection Establishment

- (CE1) I : generate nonce n_I
- (CE2) $I \rightarrow R$: I, n_I
- (CE3) R : generate nonce n_R
- (CE4) $R \rightarrow AS$: I, n_I, R, n_R
- (CE5) AS : generate key k
- (CE6) $AS \rightarrow R$: $\{\{I, n_I, R, n_R, k\}_{k_{AS}^{-1}}\}_{k_R}$
- (CE7) $R \rightarrow I$: $\{\{I, n_I, R, n_R, k\}_{k_{AS}^{-1}}\}_{k_I}, \{n_I, n_R\}_k$
- (CE8) $I \rightarrow R$: $\{n_R\}_k$

Connection Release

- (CR1) $I \rightarrow R$: $\{I, n_I, R, n_R\}_k$
- (CR2) $R \rightarrow I$: $\{n_I, n_R\}_k$

Figure 6: Underlying Authentication Protocol

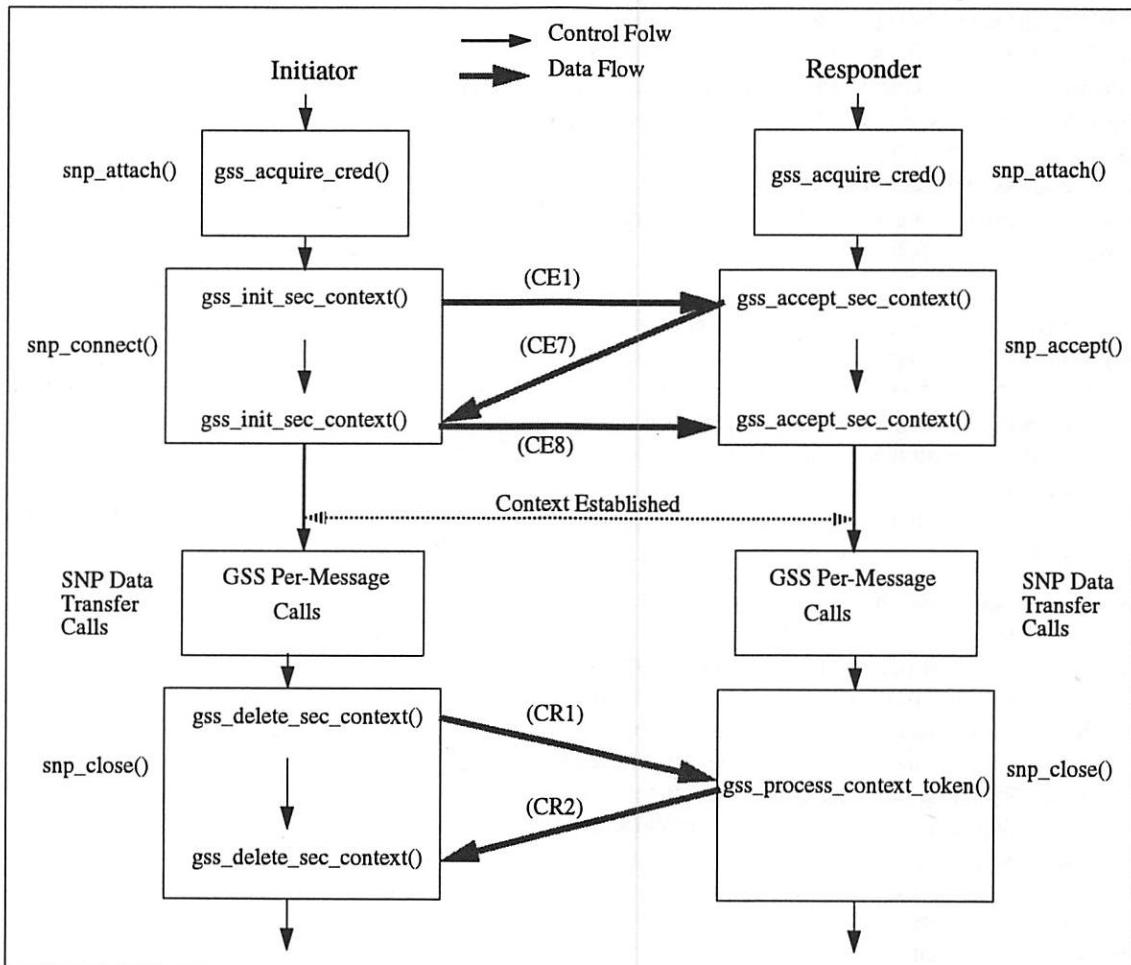


Figure 7: Control and Data Flow

SNP-API error codes as well. `snp_getpeerid()` retrieves the authenticated identity of the peer.⁹

6 Overview of Implementation

⁹ In fact, the unauthenticated identity of the peer is available as soon as the underlying authentication protocol has proceeded beyond a certain point, even if the authentication exchange fails at the end.

To facilitate discussion of SNP's implementation, it is helpful to first briefly describe our implementation of GSS-API. The authentication protocol underlying our GSS-API

implementation is shown in Figure 6 (*I* denotes the *initiator*, *R* the *responder* and *AS* the authentication server). The protocol was initially published in [22], and later verified in [20, 23]. The mapping of this protocol to GSS-API is quite straightforward, and is described in [23]. The key point to note is that the communications with *AS* (steps (CE4)–(CE6)) are completely encapsulated within GSS-API, and are not observable by the SNP layer.

Typically, an SNP-API call is translated into a number of GSS-API calls together with calls to the communication layer. GSS-API is responsible for generating *tokens* that are to be shipped using the communication layer. In simple terms, the main responsibility of the SNP layer is to request the right tokens to be generated (according to user request and current state) and to ensure they are properly transferred to the peer SNP layer. Figure 7 shows the relationship between SNP-API calls and GSS-API calls. (The bold arrows in Figure 7 correspond to the protocol steps in Figure 6.) For example, a call to `snp_connect()` results in two calls to `gss_init_sec_context()` as well as three calls to the communication layer.

There are several major considerations in implementing SNP. We describe them below:

- Two types of messages, namely, *data* and *control*, are transferred between SNP peers. Data messages contain user data and correspond to SNP data transfer calls, while control messages contain information related to the operation of the SNP layer (e.g., connection establishment request/response) and correspond to SNP control calls (e.g., `snp_connect()`) and functions (e.g., context renegotiation).

There are two ways these messages can be transferred. One is to multiplex them onto a single connection, and the other is to create dedicated connections for each type of messages. We opted for the latter because control messages should generally be given priority over data messages. Thus, if they are to be transferred on the same connection, the underlying communication mechanism must support some form of *priority* message facility. Most existing communication mechanisms (sockets in particular) do not support such priority message processing well.¹⁰ The two-connections solution avoids the dependence on such a mechanism.¹¹

¹⁰TCP does not support *out-of-band* data. It does support some elementary form of urgent data with the *urgent bit* and the *urgent pointer*. Berkeley socket supports out-of-band data, though the precise semantic guarantee is highly implementation-dependent.

¹¹In some sense, this is arguable because typically, there is no guarantee on the relative arrival times of messages sent on different connections. However, in practice, for connections with the same source and destination, the times of arrival closely follow the times of the respective sends.

- The two-connections solution also simplifies buffering concerns. Specifically, by always reading from the control connection (and responding to it) first, we no longer need to buffer all the user data preceding a control message if a control action is needed. The elimination of extra buffering also improves performance.

- The use of two connections raises the question of the address to which the second connection should be bound. Our current implementation always establishes the second (i.e., control) connection at a fixed offset from the user supplied (i.e., data) connection address. If adopted as a convention, this should not create any collision problem.

The main data structure in the SNP layer is the `snp_struct` structure. Its definition is shown in Figure 9. The `control_sockfd` and `data_sockfd` fields contain, respectively, the socket descriptors for the control and data connections. The fields `cred_list_ptr` and `ctx_list_ptr` contain pointers to GSS layer structures (see Figure 8). The meanings of most other fields are given in the comments. Each call to `snp()` creates an `snp_struct` structure; the SNP handle returned is an index into an internal table of pointers to `snp_struct` maintained by the SNP layer.

We have only touched upon the main ideas in our implementation. Most of the details concerning context expiration, context renegotiation, etc., have been omitted due to length limitation. This paper is intended only as a preliminary overview. We hope to provide a full account in a final report.

7 Performance

In this section, we present some performance results of our SNP implementation. The measurements were done on a network of Sun SPARCstations 10/30 running SunOS 4.1.3. The resolution of the system clock is in the order of microseconds.¹²

We first calibrate the performance of our cryptographic packages. Our DES package is a generic public domain one, while our RSA/MD5 package is from RSAREF [4]. Both packages are relatively portable, and are not optimized. The calibration allows us to determine the overhead introduced by the SNP layer, excluding cryptographic cost. This provides a better measure of the performance of our SNP implementation, because as more highly optimized cryptographic packages and hardware become available, the cryptographic cost will diminish, while the SNP overhead remains constant.

¹²The measurement error, however, is much worse because of context switching, function call overhead, etc.

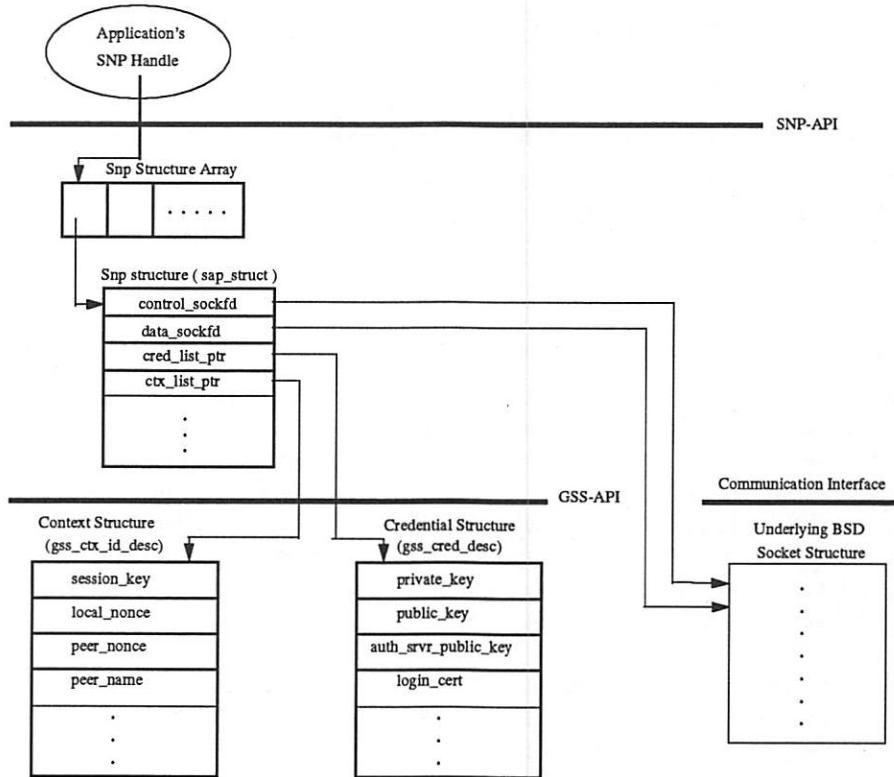


Figure 8: Data Structures

```

struct.snp_struct {
    int control_sockfd; /* Control socket desc */
    int data_sockfd; /* Data socket desc */
    int family; /* Params specified in call */
    int type;
    int protocol;
    struct.sockaddr *local_addr; /* Obtained from.snp_bind() */
    int local_addr_len;
    struct.sockaddr *peer_addr; /* Obtained from.snp_connect */
    int peer_addr_len; /* or first data xfer calls */
    struct.name_s *local_name; /* Obtained from.snp_attach() */
    struct.name_s *peer_name;
    gss_cred_id_t cred_list_ptr; /* Credentials pointer */
    ctx_list_ptr; /* Context pointer */
    int secure_options; /* Obtained from.snp_setopt() */
    int no_send; /* Options for.snp_shutdown() */
    int no_recv;
    struct.msg_s *remaining_data; /* Data recd but not requested */
    a_uint16 seq_number; /* For the GSS sequencing */
    a_uint16 recd_seq_number;
}

```

Figure 9: SNP Structure

Data Length	16B	512B	1KB	2KB	4KB	8KB	16KB	32KB
DES Encryption	0.42	2.85	5.25	9.97	19.15	37.47	75.19	152.58
DES Decryption	0.41	2.94	5.40	10.19	19.54	38.20	77.24	158.78
DES Sign	0.36	0.54	0.73	1.14	1.89	3.46	6.61	12.72
DES Verify	0.33	0.54	0.73	1.11	1.89	3.42	6.57	12.75
RSA 512 Encryption	541.24	542.21	546.00	551.92	560.44	577.69	618.54	689.76
RSA 512 Decryption	53.93	56.85	59.11	63.83	73.14	91.29	127.49	198.42
RSA 512 Sign	540.25	540.10	540.45	540.64	544.82	544.01	550.25	551.17
RSA 512 Verify	53.86	53.82	54.20	54.49	55.48	57.06	60.10	66.21
MD5	0.056	0.25	0.43	0.79	1.52	3.00	6.00	12.11

Table 1: Cryptographic Performance (in milliseconds)

		Connect		Close	
		Number of Operations	Subtotal	Number of Operations	Subtotal
RSA	Encryption	2@1kB	1092.00		
	Decryption	2@1kB	118.22		
	Sign	1@1kB	540.45		
	Verify	2@1kB	108.40		
DES	Encryption	2@200B	2.85	2@200B	2.85
	Decryption	1@200B	1.45	2@200B	2.90
	Key Gen.	1	0.38		
XDR	Encode	21@600B	29.82	2@200B	2.84
	Decode	20@600B	2.00	2@200B	0.20
Total Time			1895.57		8.79
Measured Time			2148.40		24.90
SNP Overhead			252.83		16.11
Socket Time			2.40		0.40

Table 2: Connect and Close Calls (in milliseconds)

Referring to Table 1, the following observations can be made: (1) The performance of both DES (CBC mode) and MD5 is linear with respect to data size. (2) The performance of RSA is also linear except for small data sizes. This is due to the fact that for large data sizes, the RSA implementation does not perform “true” RSA encryption. Instead, it first generates a random DES key, then encrypts the data with the DES key, and finally encrypts the DES key using RSA.

Our measurements of SNP performance are given in Tables 2 and 3. All measurements are for SNP_STREAM; similar measurements apply to SNP_DGRAM, and are omitted. Note also that these measurements are based on the use of 512-bit RSA keys (i.e., modulus).¹³

Table 2 shows the timing results for connection establishment (i.e., `snp_connect()`/`snp_accept()`) and release (i.e., `snp_close()`). The Total Time row gives the amount of time accounted for by cryptographic and XDR operations. The Measured Time row gives the observed times in establishing and closing an SNP connection. The difference between Measured Time and Total Time (the SNP Overhead row) gives the overhead introduced by SNP. The Regular Socket row gives the time it takes for the corresponding socket calls to complete. Thus, for connection establishment, SNP introduced around 0.2s overhead. A major component of this overhead is the extra round-trip delay for the communication with the authentication server and the associated message processing at the authentication server. For connection release, the SNP overhead is around 16ms.

Table 3 shows the timing results for data transfer calls (specifically for `snp_write()`). The first two rows give the times for a SNP_STREAM connection with the encrypt and sign options set, respectively. The third row gives the time for a regular SNP_STREAM with no option set, whereas the fourth row gives the time using plain sockets. The SNP Overhead row gives the overhead introduced by SNP. It can be observed that the SNP overhead is minimal.

¹³Our implementation is parametric with respect to key length. We can easily switch over to 1024-bit keys. That, however, will slow things down significantly. The increase in cost is not linear in key length.

Buffer Length	1kB	2kB	4kB	8kB	16kB
Socket SOCK_STREAM	0.6	1.0	1.3	2.8	5.2
SNP SOCK_STREAM	0.7	1.1	1.5	3.4	6.7
SNP Overhead	0.1	0.1	0.2	0.6	1.5
SNP_STREAM Plain	2.2	3.2	4.4	7.1	12.6
SNP_STREAM Signed	4.2	5.8	8.8	14.6	27.1
SNP_STREAM Encrypted	13.0	22.9	42.7	82.8	163.6

Table 3: Data Transfer Calls (in milliseconds)

Two conclusions can be drawn from these measurements: (1) The cost of cryptographic operations dominates the total cost of SNP. We believe this can be generalized to any cryptographic security mechanism. (2) It is possible to provide security at the application layer without incurring undue overhead, even with an unoptimized implementation. We expect a streamlined implementation to perform even better.

8 Related Work

Most existing work on secure network communication is focused on the protocol or architecture aspects [3, 9, 15, 17]; not much has been done concerning a general secure application network programming interface.

The work most relevant to ours includes several secure RPC systems: the secure RPC package in [2], Sun secure RPC [18] and DCE secure RPC [14]. The goals of these systems are similar to ours: to provide applications transparent access to secure communication. However, the models of communication adopted are different. RPC assumes an *implicit* communication model. That is, its users do not directly manage communications, but instead they deal with high-level abstractions in terms of procedures. SNP assumes an *explicit* communication model; SNP users are directly responsible for initiating connections, sending and receiving data, and closing connections. The same difference exists between sockets/TLI and RPC styles of network programming.

Apart from this, the implementation of these RPC systems is totally different from ours. For example, they tend to be tightly coupled to the underlying protocol (e.g., a modified Needham-Schroeder protocol [11] is used in [2], Kerberos is used in DCE). Our use of GSS-API provides protocol independence.

A recent paper by Wobber *et al.* [19] describes an operating system interface for supporting authentication. The interface is based on a formal theory of a *speaks for* relation [7]. Its concrete implementation contains several interesting abstract datatypes, e.g., a `Prin` type that represents

principals, and an *Auth* type that represents principals a process can speak for. In relating to our work, their interface can be used as an alternate lower interface for SNP. In other words, instead of translating SNP-API calls to GSS-API calls, they can be translated to calls to the interface in [19]. Such a translation should be quite straightforward because of the high level of abstraction supported. A major disadvantage of their interface, though, is the lack of compatibility with other security mechanisms, e.g., Kerberos. Moreover, their interface has only been implemented on the Taos operating system, and is currently not available on Unix.

9 Discussion and Future Work

We believe SNP represents an important first step toward secure network programming for the masses. It is clear that many important issues need to be resolved before this could be a reality. Some of these issues are: the development of a *security infrastructure* that provides uniform management and distribution of credentials (particularly for interdomain authentication), and operating system support for basic security concepts such as identity (see [19]).

One of the other impediments is performance. With rapidly improving cryptographic software and hardware, this should be a diminishing problem. As demonstrated in [5], the speed of a modern RISC-based workstation is already quite adequate for most cryptographic computation, provided the right algorithms and optimizations are used.

We are also considering several interesting extensions to the SNP interface. First, delegation can be added. This would involve the addition of two new calls: *snp_delegate()* for the delegating process and *snp_assume()* for the delegate. Delegation allows a delegate to act with the same authority as the delegating process. Second, the *snp_attach()* call can be extended to accept *identity expressions* instead of just simple identity specifications. An identity expression can specify a combination of identities that would be communicated to the peer.

In terms of implementation, we may try to port SNP to other authentication systems conforming to GSS-API. Also, the essential ideas of SNP can be adapted to provide security at other layers (e.g., transport). The lessons we learned in designing and implementing SNP provide useful references in such an effort.

Concerning the design of our interface, we have made the compatibility with sockets as one of our top design requirements. With our present design, a typical socket program can be converted into an SNP program by simply adding an *snp_attach()* call,¹⁴ without significantly modifying any of the existing code. Alternate

designs with less compatibility are possible. For example, the concept of identity can be promoted to “first class citizen” status, replacing completely the use of socket addresses. For example, the functions *snp_connect()* and *snp_accept()* would then become

```
int.snp_connect ( int.snp_ep,
                  struct.name_s.*peer_name );
int.snp_accept ( int.snp_ep,
                  struct.name_s.*peer_name );
```

Another concern in the interface design is *user control*. How much control should a user be given and how should it be done? For example, users (with the help of an operating system) may wish to explicitly manage credentials themselves, or to use their own encryption keys or algorithms. Our present design allows very limited user control (mainly through *snp_setopt()*); this could be appropriately extended.

Finally, there is the question of what the best layer for providing security support for network communication is. It can be argued that there is no single best layer for this purpose. The question then becomes: what is the best placement of security functionalities into different layers so that the resulting architecture is most general and admits least duplication? Much more research is needed to obtain an answer.

Acknowledgments

We wish to thank our shepherd Adam Moskowitz and the anonymous referees for their constructive comments and suggestions. We are also grateful to Dinesh Das and members of the Network Research Seminar of the University of Texas at Austin for listening to our ideas.

References

- [1] CCITT Recommendation X.208 Specification of Abstract Syntax Notation one (ASN.1), 1988. See also ISO/IEC 8824, 1989.
- [2] A.D. Birrell. Secure communication using remote procedure calls. *ACM Transactions on Computer Systems*, 3(1):1–14, February 1985.
- [3] J. Ioannidis and M. Blaze. The architecture and implementation of network-layer security under unix. In *Proceedings of 4th Usenix Unix Security Workshop*, Santa Clara, California, October 4–6 1993.
- [4] RSA Laboratories. RSAREF: A cryptographic toolkit for privacy-enhanced mail. January 5 1993.
- [5] J.B. Lacy, D.P. Mitchell, and W.M. Schell. Cryptolib: Cryptography in software. In *Proceedings of Usenix Unix Security Workshop IV*, pages 1–17, Santa Clara, California, October 4–6 1993.

¹⁴ And also prefixing socket calls with “snp.”.

- [6] B. Lampson, M. Abadi, M. Burrows, and T. Wobber. Authentication in distributed systems: Theory and practice. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 165–182, Asilomar Conference Center, Pacific Grove, California, October 13–16 1991.
- [7] B. Lampson, M. Abadi, M. Burrows, and T. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992. A preliminary version of this paper appeared as [6].
- [8] J. Linn. *Generic Security Service Application Program Interface*, September 1993. RFC 1508.
- [9] R. Molva, G. Tsudik, E. Van Herreweghen, and S. Zatti. *KryptoKnight* authentication and key distribution system. In *Proceedings of 2nd European Symposium on Research in Computer Security*, pages 155–174, Toulouse, France, November 23–25 1992. Springer Verlag.
- [10] National Bureau of Standards, Washington, D.C. *Data Encryption Standard FIPS Pub 46*, January 15 1977.
- [11] R.M. Needham and M.D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.
- [12] R. Rivest. *The MD5 Message-Digest Algorithm*, April 1992. RFC 1321.
- [13] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.
- [14] W. Rosenberry, D. Kenny, and G. Fisher. *Understanding DCE*. O'Reilly & Associates, Inc., 1992.
- [15] J.G. Steiner, C. Neuman, and J.I. Schiller. *Kerberos*: An authentication service for open network systems. In *Proceedings of USENIX Winter Conference*, pages 191–202, Dallas, TX, February 1988.
- [16] Sun Microsystems, Inc. *XDR: External Data Representation Standard*, June 1987. RFC 1057.
- [17] J.J. Tardo and K. Alagappan. SPX: Global authentication using public key certificates. In *Proceedings of 12th IEEE Symposium on Research in Security and Privacy*, pages 232–244, Oakland, California, May 20–22 1991.
- [18] B. Taylor and D. Goldberg. Secure networking in the Sun environment. In *Proceedings of Summer Usenix Conference*, pages 28–37, Atlanta, Georgia, June 1986.
- [19] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. In *Proceedings of 14th ACM Symposium on Operating Systems Principles*, Ashville, North Carolina, 1993.
- [20] T.Y.C. Woo. *Authentication and Authorization in Distributed Systems*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, May 1994.
- [21] T.Y.C. Woo and S.S. Lam. Authentication for distributed systems. *Computer*, 25(1):39–52, January 1992.
- [22] T.Y.C. Woo and S.S. Lam. “Authentication” revisited. *Computer*, 25(3):10, March 1992.
- [23] T.Y.C. Woo and S.S. Lam. Design, verification, and implementation of an authentication protocol. Technical Report TR 93-31, Department of Computer Sciences, The University of Texas at Austin, November 1993.

Thomas Y.C. Woo is a Ph.D. candidate at the Department of Computer Sciences at the University of Texas at Austin. His research interests include computer networking, distributed systems, security and multimedia.

Thomas received a BS (First-Class Honors) degree in computer science from the University of Hong Kong and an MS degree in computer science from the University of Texas at Austin.

Raghuram Bindignavle is a Masters candidate at the Department of Computer Sciences at the University of Texas at Austin. He received his BE in computer science at the Regional Engineering College of the University of Allahabad, India.

Shaowen Su is a graduate student at the Department of Computer Sciences at the University of Texas at Austin. He received his MS in Engineering Physics from the University of Oklahoma and his BS in Physics from Beijing University, Beijing, P.R.China.

Simon S. Lam is chairman of the Department of Computer Sciences, University of Texas at Austin, and holds two endowed professorships. Prior to joining the University of Texas at Austin faculty in 1977, he was a research staff member at the IBM T.J. Watson Research Center, Yorktown Heights, New York from 1974 to 1977. His research interests are in the areas of computer networks, communication protocols, performance evaluation, formal verification, and network security.

Simon received the BSEE degree with Distinction from Washington State University in 1969, and the MS and Ph.D. degrees in engineering from the University of California at Los Angeles in 1970 and 1974, respectively. He was a recipient of the 1975 Leonard G. Abraham Prize Paper Award from the IEEE Communications Society. He organized and was program chair of the first ACM SIGCOMM Symposium held at the University of Texas at Austin in 1983. He presently serves on the editorial boards of *IEEE Transactions on Software Engineering* and *IEEE/ACM Transactions on Networking*.

An Efficient Kernel-Based Implementation of POSIX Threads

Robert A. Alfieri
Data General Corporation

Abstract

This paper describes the kernel-based implementation of POSIX Threads (Pthreads) in the DG/UX™ operating system. The implementation achieves time efficiency by using a general-purpose trap mechanism, known as a *Kernel Function Call* (KFC), that carries an order of magnitude less overhead than a traditional system call. On a 50 MHz Motorola MC88110, the implementation can create and exit a thread (with the associated context switch) in 8.1 microseconds and yield to another thread in 4.0 microseconds. The implementation also achieves space efficiency by paging and decoupling bulky data structures.

The advantages of a kernel-based implementation include design simplicity, less code redundancy, optimization of global (interprocess) operations, avoidance of inopportune preemption, and global semantic flexibility. The disadvantage is a monolithic design that lacks user-level flexibility.

1. Introduction

Threads provide an efficient and convenient concurrent programming paradigm for applications running on shared-memory multiprocessors. As industry-standard thread interfaces such as POSIX Threads (Pthreads) [Pos93] find their way into open systems, an increasing number of portable applications are being written (or rewritten) to exploit threads.

Support for Draft 6 of Pthreads was shipped in version 5.4R3.00 of DG/UX™, Data General's commercial UNIX® operating system. DG/UX originated in 1984 as a rewrite of the UNIX kernel in order to support Symmetric Multiprocessing (SMP), full preemption, and better modularity [Kel89].

Though this paper focuses on the techniques that DG/UX uses to implement threads efficiently in the kernel, any operating system could apply the same techniques to other thread packages or performance-critical system calls. Most importantly, none of these techniques are specific to DG/UX or POSIX Threads.

2. Design Overview

The overriding design goal is to allow standard multithreaded applications to map as efficiently as possible (both in terms of time and space) onto current and future multiprocessor architectures. In order to realize this goal, the implementation employs three main features:

Kernel Function Calls (KFCs)

KFCs are fast, general-purpose kernel traps that allow Pthreads to be implemented simply and efficiently in the kernel.

The implementation uses the same set of KFCs to optimize both local (intraprocess) and global (interprocess) thread operations. Global operations are important today and will become increasingly important as applications and machines move towards a paradigm of distributed computing.

Low Memory Overhead

By keeping the per-thread physical and virtual memory consumption to a minimum, the implementation can easily support thousands of threads in a single process.

All per-thread data structures are pageable, except a 128-byte kernel-level structure. Moreover, space-consuming kernel stacks and other *transient data* are tied only to threads that have entered the kernel for a traditional system call, page fault, or other type of full kernel entry.

Pthread Groups and Hierarchical CPU Affinity

These extensions give applications explicit control over the scheduling of threads onto the underlying multiprocessor machine.

The affinity mechanism is hierarchical in that it parallels the CPU/cache/memory hierarchy of the underlying machine. Applications may group together threads working on the same data set, then affix that thread group to a set of CPUs sharing the same cache or local memory. This feature is particularly important for improving cache locality on large SMP or NUMA (Non-Uniform Memory Access) machines. If an application does not specify its thread grouping or affinity, the operating system performs automatic thread grouping and affinity assignments.

The rest of this paper is devoted to describing the motivations and details of the implementation with respect to the first two features. The last feature is described separately [Alf94].

3. Application Requirements

There are two broad classes of thread operations:

- 1) *Local Operations* are those operations that occur among threads in the same process. Local operations do not involve communication with other processes and are often referred to as *intraprocess* or *intra-address-space* operations.
- 2) *Global Operations* are those operations that occur between threads in two different processes or between a thread and the kernel. Global operations are often referred to as *interprocess* or *inter-address-space* operations.

A thread implementation should optimize both types of operations. Because global operations require more work, they tend to be somewhat slower than local operations. However, a request should only incur a cost that is proportional to the service that must be delivered.

3.1. Local Operations

Ideally, a threaded application spends all of its time in user space performing local operations and rarely talks to the kernel or to other processes. Examples of using local-only operations include situations in which

- *Database* back-end threads perform a parallel sort of a database table stored in process memory.

- *Real-time* threads are used to prioritize certain events that are handled entirely in user space.
- *Simulation software* breaks up a problem into smaller simulations that run as threads on separate CPUs.

3.2. Global Operations

Commercial applications also make extensive use of global operations. For data integrity reasons, applications are split into multiple processes, and shared system services reside in the kernel or in privileged server processes. The kernel must always participate in (safe) global thread operations, as illustrated by the following examples in which

- *Database* front-end threads go through the kernel to talk to database back-end threads which, in turn, make kernel system calls to transfer data to or from permanent storage.
- *Real-time* threads require kernel-based global scheduling in order to minimize response time to global asynchronous events.
- *Distributed computing* services are divided into replicated multithreaded processes, which communicate using a *Local or Remote Procedure Call (RPC)* mechanism.

Clearly, global interactions play an important role in current and future multithreaded applications. Implementations should optimize both local and global cases.

4. Previous Work

This section introduces library-based implementations of threads and explains why DG/UX has not adopted the same approach. Next, the discussion focuses on previous work within DG/UX that ultimately led to an efficient kernel-based implementation.

4.1. Multiplexed Libraries

In pursuit of optimal performance, numerous multiplexed thread libraries have been developed to avoid kernel system calls, especially during performance critical operations such as thread creation and context switching [Gol90, Pow91, Ste92, Mue93].

Multiplexed libraries employ two levels of scheduling. User-level threads are multiplexed onto a typically smaller number of kernel-level entities (e.g., processes or kernel-level threads). In the simplest

implementations, all user-level threads are multiplexed onto a single process. In more sophisticated implementations, the number of kernel-level entities varies with the number of CPUs that are assigned to the process. Context switches among kernel-level entities involve slower system calls. Context switches among user-level threads occur entirely in user space, thus optimizing performance. In addition, flexible multiplexed libraries [And91b, Mar91] allow applications to integrate their own thread schedulers.

Because multiplexed libraries reside in user space, highly tuned library primitives can be used only for local thread operations. Global thread operations, such as interprocess synchronization and RPCs, typically follow significantly slower system call paths or use dedicated kernel-entry mechanisms.

In addition, well-integrated multiplexed libraries require complex algorithms to bridge the wide gap between library and kernel databases. In particular, the library and kernel must continuously inform each other of the number of runnable threads in user space and the number of available CPUs in the kernel. Refer to the section on "Kernel vs. User Threads" for a more detailed discussion of this and other issues.

Despite their complexity, multiplexed libraries prevail due to the perception that system call overhead would dominate the cost of a kernel-based thread primitive.

4.2. System Call Overhead

This section explains in greater detail why multiplexed library-based systems and DG/UX have avoided using traditional system calls to implement threads.

Have System Calls Gotten Relatively Slower?

Recent research in the area of RISC processor performance [Ous90, And91a] asserts that operating system primitives such as system calls, exceptions, and process context switches have not experienced the same accelerated speedup as application integer and floating-point operations. There are two reasons why this observation may hold true for system calls in many commercial operating systems:

- 1) System calls are typically more memory-intensive than arithmetic operations, thereby introducing more processor stalls in the form of cache misses and degraded instruction-level parallelism.

- 2) System call entry has grown more complex due to the introduction of additional functionality and complexities in the kernel.

As an example, DG/UX system call overhead has increased mainly for the following reasons:

- 1) RISC processors have required the kernel to save and restore more registers and to do more work during system calls than previous CISC processors.
- 2) The desire to provide more precise execution time accounting has resulted in reading hardware timers during system call entry and exit. Historically, time accounting has been implemented using a coarse ten-millisecond clock tick.
- 3) An effort to make the kernel more modular and portable has introduced more inter-subsystem calls during the system call path.

Reasons (1) and (3) likely apply to many other operating systems, while reason (2) may be specific only to DG/UX and a few other commercial systems.

Regardless of the actual implementation, system call overhead is inherently significant due to user state saving and restoring, and preparing the caller for execution in a fully preemptive SMP kernel.

Impact on Thread Primitives

Most local thread primitives execute in microseconds or tens of microseconds. System call overhead, which is on the order of tens of microseconds, would contribute significantly to the overall cost of a kernel-based thread primitive. Roughly speaking, a system call imposes the equivalent of (at least) an additional thread-to-thread context switch on each kernel entry.

4.3. Down-Sizing System Calls

Fortunately, system calls are not the only way to enter the kernel. Modern RISC and CISC processors provide fast trap instructions, which can be used to invoke simpler kernel primitives that are an order of magnitude faster than traditional system calls.

For example, on a (relatively slow) 25 MHz Motorola MC88100 [Mot91], the cost of trapping into the kernel and returning from the trap, including proper Processor Status Register (PSR) and Program Counter (PC) setup and restoration, is less than one microsecond. A null system call on the same machine takes approximately 20 microseconds.

The actual trap into the kernel is inexpensive. The real issue is the costly steps that must be performed in order to set up for a traditional system call. In fact, most of these steps can be avoided by using a simpler primitive.

4.4. DG/UX Extended Operations (XOPs)

For a number of years, DG/UX has used fast kernel traps to implement certain extended operations (XOPs) that are not provided directly by the Motorola 88000 instruction set, such as atomic-increment and conditional-store to user memory locations. Though these atomic operations could have been implemented completely in user space using a lock based on the 88000's test-and-set instruction, there was a desire to avoid any unbounded delay associated with *inopportune preemption* (e.g., hardware interrupt, page fault, process abort) while holding a user-level spin lock or mutex.

XOPs prevent this inopportune preemption by using simple traps across the user-kernel protection boundary. A null XOP takes less than one microsecond and scales directly with processor speed. Atomicity among multiple CPUs is provided using a spin lock inside the kernel. In the 88000 family, there is no need to disable CPU interrupts explicitly because they are implicitly disabled by the trap. Thus, preemption is avoided while executing an XOP and holding a spin lock.

Once the XOP has acquired the spin lock, the XOP is free to perform the atomic increment or conditional store on the user memory location. However, this memory location could be paged out or invalid. Because the XOP holds a critical spin lock and has not fully entered the kernel, it must be prepared to deal with a fault when touching the user memory location. To this end, a flag is set in per-CPU data indicating that an XOP is in progress. If a fault occurs while touching user memory, the fault prehandler ignores the fault and backs out of the XOP. The failure is reported back to the user-space library routine, which is then responsible for touching the memory location. Touching the memory location causes a normal page fault to bring in the memory page. Then the library routine retries the XOP; the XOP almost always succeeds on the second try.

4.5. Fast Traps in Other Systems

Fast traps have been used in numerous other operating systems. In particular, fast traps have been successfully exploited for the purpose of speeding up

Interprocess Communication (IPC) [Ber90, Lie92, Wal92, Lie93]. Like XOPs, these traps are specific to the primitive that they implement.

5. Kernel Function Calls (KFCs)

The powerful combination of fast kernel trap and fault interception used in XOPs is at the heart of the kernel-based thread implementation. However, XOPs do not provide the level of semantic flexibility and ease-of-use that thread primitives require. In particular, XOPs must be implemented in assembly language, the XOP back-out mechanism is cumbersome, and XOPs do not check for preemptions before leaving the kernel. For these reasons, XOPs were abandoned in favor of a more general and usable flavor of fast kernel trap, known as a Kernel Function Call (KFC).

5.1. KFC Semantics

As the name implies, a KFC is a function that resides in the kernel and is callable from user space. Calls to KFCs follow the same parameter passing and register saving conventions of the underlying processor architecture. Returns from KFCs follow the same conventions as UNIX system calls on the underlying processor architecture. KFCs carry more overhead than an XOP, but still an order of magnitude less overhead than a traditional system call. Actual KFC overhead is on the order of 20-30 machine instructions and scales directly with processor speed.

KFCs operate in a restricted environment where CPU interrupts are disabled, time is still charged to user space, and blocking with a kernel stack is not permitted. Although these restrictions could be lifted in order to implement all system calls using KFCs, only the shortest system calls would benefit noticeably. Also, complete generality would bloat KFC overhead and introduce undesirable complexities, such as the need to unwind a KFC so that a user debugger could manipulate a thread's user registers. For these reasons, as well as the desire to avoid other likely additions to the KFC entry and exit paths, DG/UX limits KFC usage to performance-critical primitives (e.g. Pthreads) that can operate in this restricted environment. Other potential uses include `getpid()`, `gettimeofday()`, and RPCs.

Appendix A gives a complete list of thread-related KFC interfaces. Table 1 illustrates the semantic differences among XOPs, KFCs, and system calls. Note that numerous costly steps are avoided during KFCs and XOPs.

5.2. KFC Details

The following sections describe the general KFC mechanism in more detail, with emphasis on using KFCs to implement thread primitives. The reader may wish to browse or skip these details, then return to them after reading the rest of this paper.

5.2.1. KFC Entry

The steps in KFC entry proceed as follows for all KFCs:

- 1) A thread makes a call to a library routine.
- 2) The library routine typically performs a few steps, then decides to invoke a KFC. The routine packages some arguments and traps into the kernel through a common vector.
How the arguments are packaged and how the trap is performed are architecture-dependent. For example, on RISC machines, the arguments are typically passed into the kernel via registers.
On CISC machines, the processor may automatically copy the arguments into the kernel, or it may provide enough registers to hold all of the arguments.
- 3) Once in the kernel, common KFC entry code saves the thread's user-level return address and PSR in per-CPU data.
Assuming that all KFCs are written in C, the entry code saves the user-level stack pointer and switches to a kernel-level stack. On CISC architectures, these operations may be performed automatically by the trap instruction.
- 4) Next, the entry code transfers control to the actual KFC. On RISC architectures, the arguments to the KFC are typically already in the appropriate registers. On CISC architectures, the arguments may need to be pushed onto the kernel stack before the call. However, argument pushing comes cheaply on modern CISC processors.
- 5) The actual KFC looks like a traditional system call written in C.

Aspect of Kernel Entry/Exit	X O P	K F C	S Y S C A L L
Enters the kernel using a fast trap instruction and leaves the kernel using a fast return-from-trap instruction	✓	✓	✓
Uses common entry/exit code		✓	✓
Performs a full register state save on entry and full register restore on exit			✓
Switches to system time accounting on kernel entry, then back to user time accounting on exit			✓
Prepares the environment for full kernel entry and enables interrupts			✓
Executes with CPU interrupts disabled	✓	✓	
Can be (and is typically) written in C		✓	✓
Can block the calling thread and switch to another thread		✓	✓
Frees up the kernel stack for reuse when the calling thread is blocked		✓	
Can unblock other threads and check for preemption before leaving the kernel		✓	✓
Handles faults directly without explicit detection			✓
Backs out of faults that occur while touching pageable user or kernel memory	✓	✓	
Backs out of faults and <i>promotes</i> to an internal system call to handle the fault		✓	
Follows system call return conventions		✓	✓

Table 1: Semantic Differences

5.2.2. KFC Exit

When the thread has completed its KFC, the steps in KFC exit proceed as follows for all KFCs:

- 1) From within the KFC, the thread temporarily stores its primary and secondary return values to per-CPU data variables. Note that the use of per-CPU data is allowed because CPU interrupts are disabled.
- 2) The thread returns naturally from the KFC. The return status indicates whether the KFC completed normally.
- 3) The common KFC exit code reads a different per-CPU variable to determine if the calling thread should check for priority preemption. This occurs if the thread had awakened higher priority threads during the KFC. In this case, the calling thread yields to another thread and will eventually return to this exit code.
- 4) The KFC exit code interrogates the return status from the KFC. For a normal return, the saved primary and secondary return values are returned to the library routine. The conventions for returning these values are the same as those for system calls. For a failure return, the `errno` value is extracted from the return status, and the primary and secondary return values are ignored.
- 5) The KFC exit code restores the partially saved user state, and executes the appropriate instruction(s) for returning to user space.
- 6) The library routine interrogates the results of the KFC in the same way that a library routine would interrogate the results of a system call.
- 7) The library routine may perform other steps before returning to its caller.

5.2.3. Thread Reschedule

Some KFCs do more than enter the kernel, perform a few steps, and return. In numerous cases, the calling thread must suspend its execution. Examples of thread suspension include the following:

- The calling thread attempts to *join* (wait for) a thread that has not yet terminated.
- The calling thread explicitly *yields* to another thread of equal or better priority.
- The calling thread awaits the *release* of a *mutex* or the *signaling* of a *condition variable*.
- The calling thread *sleeps* for a specific amount of time.

When a thread suspends, it unwinds the KFC that it is executing so that the thread's register state is readily accessible. Then common KFC code saves the full register state in a standard `mcontext_t` structure that resides inside the kernel. Next, the thread calls into the dispatcher, which switches control to another thread.

Eventually, the original thread will be awakened and a CPU will continue the thread's execution. The thread's *continuation function* [Dra91] restores the thread's register state from its `mcontext_t` structure and resumes the KFC where it left off.

5.2.4. KFC Fault Detection

Like XOPs, KFCs must back out of faults that occur while referencing pageable kernel or user memory. There are two main reasons for backing out:

- 1) The KFC operates in a restricted kernel environment with CPU interrupts disabled. Full fault handling is not permitted in this environment.
- 2) The KFC typically holds a critical spin lock. If the KFC were to allow the fault to be processed, the spin lock could remain held for an indefinite period of time. This could lead to deadlock or extremely long latencies for other threads in the same process. *Fault detection actually eliminates long latencies associated with critical locks that are used to implement threads.*

Faulting on Kernel and User Memory

Because `mcontext_t` structures can consume several hundred bytes, depending on the processor architecture, the implementation allows these structures to be paged. KFCs must be prepared to back out of page faults that occur while saving/restoring state to/from the per-thread `mcontext_t` structure located in the kernel.

In addition, a few thread KFCs need to reference user space from inside the kernel. For example, when a thread cannot immediately obtain a Pthread mutex in user space (i.e., failing the uncontested case), the thread invokes a KFC to await the release of the mutex. In this contested case, the KFC saves the thread's state and adds the thread to a kernel-level synchronization queue associated with the mutex. However, due to the potential for certain race conditions, the KFC must re-check the user-level mutex to see if the mutex has been released before actually putting the thread to sleep.

The KFC, while attempting to touch the mutex, must be prepared to back out of two types of faults:

- 1) The user page holding the mutex is paged out.
- 2) The user mutex address passed into the KFC no longer refers to valid user memory. Although the Pthread library dereferences the mutex address and checks the validity of the corresponding mutex structure, another thread could erroneously unmap the mutex memory before the library invokes the KFC.

Detecting Faults

Whenever a KFC needs to touch kernel or user memory that could cause a fault, the KFC calls a trivial assembly language routine to attempt the memory operation. Before performing the operation, this common routine sets a per-CPU variable to the address of back-out code at the bottom of the routine. Under normal operation, this per-CPU variable holds a value of zero, indicating that faults are not being intercepted.

As with XOPs, if a fault occurs while touching the memory location, the appropriate fault prehandler checks the per-CPU variable. Since the variable has a nonzero value, the prehandler knows not to proceed normally to the full fault handler. Instead, the prehandler ignores the fault and branches to the back-out code at the bottom of the common assembly language routine. The back-out code returns to the KFC, indicating that a failure occurred while touching the memory location. If no fault had occurred, the assembly language routine would have simply returned normally after successfully manipulating the memory location.

Unlike XOPs, whenever a KFC access to kernel or user memory fails, the KFC is *promoted* directly to an internal system call without returning to user space, as discussed in the next section.

5.2.5. KFC Promotion and Demotion

In most cases, thread operations are completed entirely at the KFC level. However, whenever something complicated occurs that cannot be handled at the KFC level, the KFC must be *promoted* to an *internal system call*. These rare complications include the following:

- 1) Faulting on user memory, such as a user-level mutex.
- 2) Page faulting on kernel memory, such as the per-thread `mcontext_t` during register state save or restore.

- 3) Failing to allocate memory resources without blocking, for example, during thread creation when no dead thread is available for “reincarnation.”
- 4) Detecting the presence of a software interrupt sent to the calling thread for such events as a signal, Pthread cancellation, abort, or stop.

In each of these cases, the cost of completing the operation is high relative to system call overhead. Therefore, neither internal system call performance nor promotion overhead is critical.

Implementation

The basic idea behind KFC promotion is to make the environment appear as if the calling thread invoked a system call instead of a KFC. The promotion proceeds as follows:

- 1) The detecting KFC stores in per-CPU data the address of its associated internal system call and parameters for the call.

Typically, most of the parameters that were passed to the KFC are also passed to the internal system call. Furthermore, because complications can occur during various *phases* of the KFC, an indication of the phase is also passed as an extra argument to the internal system call. This indicator tells the internal system call where to resume the operation, for previous phases have already been completed successfully.
- 2) The KFC returns a special status code, indicating that the KFC is being promoted to a system call. The common KFC exit code recognizes this special return status and proceeds with the promotion instead of returning from the KFC. Essentially, the KFC exit code arranges the environment so that it appears as if the calling thread has just trapped into the kernel to execute a system call. However, instead of having passed in a system call number, the thread has “passed in” the kernel address of the internal system call function. Also, unlike a failed XOP, the promoted KFC does not force the thread to return to user space; the promotion takes place completely within the kernel.
- 3) The calling thread follows the normal system call entry path, except that the system call entry code understands that this is an internal system call. In particular, the handler knows that the internal system call address has been “passed in” rather than a system call number.

- 4) The internal system call interrogates its arguments to determine the next phase that needs to be completed. Because the thread is running in a full-fledged system call, the thread is free to page fault on user memory, handle signals, etc.
- 5) Once the thread completes the slow phase, it may decide to complete other phases, suspend itself, or return from the system call.

If the thread decides to suspend itself (in the same way that it would have in the KFC), it *demotes* the system call to the KFC so that it can free up its kernel stack for reuse (refer to the section on “Transient Data”). When the thread is continued, it resumes at the KFC level as if it had never been promoted. Note that the KFC may get promoted again if it encounters a complication during a later phase.

If, on the other hand, the KFC had been promoted to handle one of the last phases of the KFC, the thread simply returns from the internal system call after completing these phases. When the thread gets back to user space, the library routine recognizes no difference because KFCs follow the same return conventions as system calls.

Note the following about KFC promotion and demotion:

- Only those KFCs that can encounter complications are promotable.
- Each promotable KFC is associated with an internal system call that mimics some of the phases of the KFC.
- Internal system calls tend to be significantly larger than their corresponding KFCs because the KFCs handle only the common, performance-critical cases. In order to reduce code redundancy, internal system calls share most of the same support code as their corresponding KFCs. With minor effort, it should be possible to condense the KFC and the internal system call into one routine.
- The system call entry code remains nearly the same for both normal system calls and internal system calls. The only differences are the specification of the actual call routine and, on some CISC architectures, where the arguments are located. The system call exit code is exactly the same for both types of system call.
- Given that promotion is rare, the only good reason for demotion is kernel stack reuse.

5.3. Performance Measurements

The following tables demonstrate that thread operations can be implemented efficiently using KFCs. All measurements were taken on a dedicated Motorola 50MHz MC88110 uniprocessor [Mot91] using Draft 6 of Pthreads as shipped in the DG/UX 5.4R3.00 operating system. Each reported measurement reflects the average elapsed time per iteration of a loop that repeatedly invokes the associated primitive(s).

Kernel Entry Overhead

Table 2 illustrates the costs of XOPs, KFCs, and system calls. All calls include the overhead of a user-level “wrapper” function in a shared library.

Note that the null system call time is accurate for the DG/UX kernel even though the time exceeds the number reported by research projects such as [And91a]. This discrepancy is explained by the additional steps that DG/UX takes in order to support SMP, precise time accounting, multiple APIs, and greater internal code modularity.

Kernel Entry Type	Time (usec)
Null XOP (basic trap overhead)	0.5
Null KFC (written in C)	1.4
Null System Call	19.5

Table 2: Kernel Entry Overhead

Local Operation Overhead

Table 3 gives times for local (intraprocess) thread operations. As in library-based systems, local threads are the fastest because they share the same CPU time accounting (and timeslice) and get scheduled onto process-local queues. During local context switches, the implementation can thus avoid time bookkeeping and global scheduling decisions.

In all cases, local thread operations outperform the null system call. The breakdown for thread create/exit was determined using a hardware logic analyzer

Local Operation	Time (usec)
Null thread yield (no context switch)	1.8
Thread Yield (with context switch)	4.0
Thread create/exit (with context switch) Breakdown:	8.1
• create (with KFC entry/exit)	4.0
• exit (with KFC entry)	1.6
• context switch (with KFC exit)	2.1
• invoke new thread's start routine	0.4
Contested mutex lock/unlock (with context switch)	12.3
Condition wait/signal (with context switch)	13.2
Thread create/exit/join (with both context switches)	14.1
Timed condition wait/signal (with context switch and one-second time-out overhead)	17.7

Table 3: Local Thread Operations

Task Creation Overhead

Table 4 gives the time required to create and wait for a thread or process to exit. Times are given for both locally and globally scheduled Pthreads. Unlike local threads, global threads have dedicated CPU time accounting and get scheduled onto system-wide queues. The local time has been copied from Table 3 for comparison. Total time includes the context switches to and from the new thread or process.

Create/Exit/Wait (with both context switches)	Time (usec)
Locally scheduled thread	14.1
Globally scheduled thread	57.1
Process (using fork)	7738.9

Table 4: Task Creation Overhead

Context Switch Overhead

Table 5 gives times for various types of context switches involving Pthread condition variables. The

overhead for obtaining and releasing the associated mutex is included in the condition-variable times. In Table 3, a time was given for locally scheduled threads using local (intraprocess) condition variables. Table 5 repeats the same value and includes those for other combinations of locally vs. globally scheduled threads and local vs. global condition variables. Table 5 also includes a time for two processes using a global condition variable in shared memory. Note that two processes cannot communicate via local condition variables because local condition variables are, by definition, only available within a single process. Lastly, times are included for traditional UNIX (System V IPC) semaphores in order to illustrate the advantage of using condition variables in multitasking applications.

Sleep/Wakeup Pair (with context switch)	Time (usec)
Local threads, local condition	13.2
Local threads, global condition	13.7
Global threads, local condition	33.6
Global threads, global condition	38.4
Two processes, global condition	38.9
Local threads, UNIX semaphore	99.7
Global threads, UNIX semaphore	107.9
Two processes, UNIX semaphore	116.1

Table 5: Context Switch Overhead

Comparison with Library Implementations

Table 6 compares the local-operation performance of DG/UX threads with two library implementations that were presented at previous USENIX conferences. The first is a pure library implementation from Florida State University [Mue93], which reported best numbers for a 40MHz SPARC™ IPX. The second is the SunOS™ multiplexed implementation running on a 25MHz SPARC 1+ [Pow91]. Given the disparity in processor types and the fact that thread implementations undergo continual tuning, Table 6 does not attempt to declare one implementation superior to the others. Nonetheless, if one scales the measurements in Table 6 by processor speed, one sees that kernel-based implementations can perform local Pthread operations in roughly the same time as library-based implementations.

Local Operation	DG/UX 50MHz Moto 88110 (usec)	FSU 40MHz SPARC IPX (usec)	SunOS 25MHz SPARC 1+ (usec)
Thread Create (no context switch)	4.0	12.0	56.0
Mutex Lock/Unlock (with context switch)	12.3	51.0	not given
Condition wait/signal (with context switch)	13.2	55.0	158.0

Table 6: Comparative Measurements (not scaled)

Summary

KFCs make both local and global thread operations efficient. In particular, the performance of local and global operations roughly matches the performance of local operations cited by previous library implementations.

6. Minimizing Memory Consumption

In order to meet the design goal of minimizing per-thread memory consumption, the implementation pages and decouples data structures wherever possible. Since DG/UX already pages kernel data, the paging of thread structures did not require special work in the virtual memory subsystem.

6.1. Kernel-Level Thread Structure

The only per-thread data that cannot be paged is the 128-byte main control block structure that resides in the kernel address space. An application that uses as many as 1,024 threads consumes only 128KB. Given current trends in memory capacities, this amount of overhead is insignificant. Even on a 16MB workstation, swap space for 1,024 threads (and their associated user stacks) is a more limited resource than 128KB of physical memory. On a medium-sized server (the primary design target), the impact of this 128KB is even more negligible.

6.2. User-Level Thread Structure

Every thread has a small user-level structure that resides at the base of the thread's stack. This structure and its enclosing thread stack are entirely pageable. The following information is stored in this

structure:

- a cached copy of the thread's ID
- the per-thread errno variable
- a pointer to the thread's most recently pushed Pthread cleanup handler
- the values for the Pthread thread-specific data variables

6.3. Kernel-Level mcontext_t

Every thread has a kernel-level `mcontext_t` structure that is used to store the thread's register state when it is suspended in a KFC. This standard UNIX structure can consume several hundred bytes and is entirely pageable. As discussed earlier, KFCs must be prepared to back out of page faults on this structure while saving or restoring register state.

6.4. Transient Data

When a thread enters the kernel for a traditional system call, hardware interrupt, page fault, or other type of exception, the thread must run on a kernel stack. The kernel stack, along with other variables that are needed for one trip inside the kernel, make up *transient data*. The size of transient data is approximately 8KB; thus, it is too expensive to provide each thread with a dedicated transient data area. In addition, transient data must be made temporarily non-pageable while a thread is running on a CPU.

In order to reduce the amount of pageable and non-pageable transient data, the implementation *decouples* transient data from threads that are no longer using it. As the word "transient" implies, transient data are needed for only one trip inside the kernel and can migrate from thread to thread. The following rules summarize transient data allocation requirements:

- When a thread is actually running on a CPU, the thread must have non-pageable transient data because the thread must be able to enter fully into the kernel for any reason (e.g. to process an exception).
- Once a thread enters fully into the kernel, its transient data remain assigned to the thread until the thread leaves the kernel or exits. However, once the thread is descheduled (e.g. while sleeping for a long time in a system call or when the system is loaded), the kernel may decide to page out the thread's transient data.

- In contrast, when a thread suspends in a KFC, it relinquishes its transient data for reuse by other threads in the process. This is possible because the thread has not entered fully into the kernel, has saved its state in its kernel-level `mcontext_t` structure, and no longer needs its kernel stack. This rule applies also to global (interprocess) operations that are implemented using KFCs.
- When a suspending thread demotes an internal system call to a KFC, the suspending thread relinquishes its transient data. Similarly, when a thread is preempted directly out of user space as a result of taking a timeslice or other hardware interrupt, the suspending thread demotes to the KFC level and frees up its transient data.
- The kernel regulates the number of transient data areas in the process based on the current number of threads that are making system calls, page faults, etc. The kernel regulates the number of non-pageable transient data areas based on system-wide load and process priorities.
- *In a typical application, it is common to have many threads sharing a small number of transient data areas.*

7. Kernel vs. User Threads

Given that threads can be implemented efficiently in the kernel, what are the advantages and disadvantages of kernel-based threads vs. library-based threads?

7.1. Design Simplicity

Kernel-based implementations are inherently simpler than multiplexed implementations because they employ only one level of thread scheduling; they need not treat both user-level threads and kernel-level entities. This eliminates the communication gap between library and kernel databases that is common in multiplexed implementations. The kernel sees all threads in the system and can simply schedule runnable threads directly onto available CPUs with minimum latency. Furthermore, because critical data structures (including sleep queues) reside in the kernel address space, an errant application cannot corrupt them.

A kernel-based implementation also reduces the amount of code redundancy found in multiplexed implementations. For example, there is no need to have two sets of user synchronization primitives, one

for user-level threads and one for kernel-level entities. The same holds true for thread creation, signal handling, thread timeslicing, user debugging support, etc.

On the other hand, many Pthread semantics creep into the kernel. If the thread library does not provide support for a particular thread package or function, the application writer may need to wait until the kernel provides the desired functionality. Naturally, many thread packages can be emulated efficiently on top of existing Pthread or KFC semantics. Nonetheless, as a kernel-based implementation accumulates more semantics, its monolithic design increases in complexity.

7.2. Performance Focus

Library-based systems focus their performance effort in the user-space thread library. Thus, highly-tuned thread primitives can be exploited only for local thread operations.

Kernel-based systems focus their performance effort in the kernel address space. Highly-tuned KFCs can be used for both local and global thread operations. For example, one KFC handles thread suspensions for both local and global condition variables. As a result, when the same type of thread is involved, the overhead for a global condition variable is only slightly greater than that for a local condition variable.

7.3. Avoiding Inopportune Preemption

Library-based implementations protect thread data structures with critical user-level locks. If a pre-emption or page fault occurs while a critical lock is held, other thread operations could be delayed for tens of milliseconds (or more) until the critical lock is finally released. Such latencies hurt performance, scale badly with increased numbers of CPUs, and can cause real-time applications to miss deadlines.

While complex schemes exist to work around this problem, a kernel-based implementation provides a natural solution. During KFCs, interrupts are disabled, so no timeslice or other preemption can occur. Furthermore, if a page fault occurs while touching user space (e.g., retrying a mutex before going to sleep), the page fault is intercepted, any critical kernel spin lock is released, and the KFC is promoted to an internal system call to handle the page fault as if it had occurred from user space.

7.4. Semantic Flexibility

Generally speaking, library-based implementations provide more local semantic flexibility, while kernel-based implementations provide more global semantic flexibility.

Unlike the most advanced multiplexed implementations [And91b, Mar91], kernel-based implementations do not allow applications to directly manipulate the internal scheduling mechanism or integrate their own scheduling policies. Though POSIX provides a rich set of basic mechanisms and policies on which more complex constructs can be built, a flexible library implementation will allow the same constructs to be expressed, while maintaining excellent integration with the rest of the library.

On the other hand, a kernel-based implementation can efficiently accommodate additional types of global semantics. For example, DG/UX allows applications to establish affinity relationships between groups of threads and sets of CPUs sharing the same cache or local memory [Alf94]. The implementation maintains local-operation performance within Pthread groups, while minimizing penalties for intergroup operations. Library-based implementations typically inflict greater penalty for CPU affinity because they require that user threads be "bound" to slower kernel-level entities.

7.5. User Tools

In a kernel-based implementation, the kernel sees all threads in the system. This simplifies and enhances the development of user tools that support threads. For example, user debuggers need only converse with the kernel, and the kernel always supplies a consistent set of information. In library-based implementations, the debugger must talk to both the kernel and the user-level library, and the library could be in an inconsistent state.

DG/UX provides a `ps` command option that displays the detailed status of all threads in the system. This status information includes, for example, the user address of the mutex or condition variable on which a thread is blocked. Most library-based implementations show information only for the underlying kernel entities, which have little correlation with library-level threads.

7.6. The Best of Both Approaches

Library-based systems could implement their kernel-level entities more efficiently using KFCs.

Such a hybrid would improve the performance of global operations, while retaining flexibility in user space.

Conversely, kernel-based systems could provide additional mechanisms found in flexible multiplexed implementations. For example, a kernel-based implementation could provide Pthread extensions that allow a parallel language runtime library to better regulate the number of active worker threads based on the current number of assigned processors.

8. Conclusions

This work demonstrates how challenging basic assumptions (e.g., that threads cannot be implemented efficiently using system calls) can favorably redirect the course of a design. This work also illustrates the trade-offs involved in implementing operating system primitives in kernel space vs. user space.

For example, when KFCs are combined with the conservative use of memory, the result is a simple and efficient kernel-based implementation of POSIX Threads. While such an implementation forfeits some semantic flexibility in user space, it optimizes both local and global operations, and can easily accommodate additional global semantics.

Finally, any operating system could use the same general KFC mechanism to implement its own thread package or other types of fast kernel primitives, including the following global operations:

- Local or remote procedure call
- Global asynchronous event notification in the form of a newly created thread
- Interprocess thread creation or migration
- Quick `gettimeofday()`, `getpid()`, etc.

9. Acknowledgments

Eric Hamilton and Michael Kelley developed XOPs. Steve Daniel suggested using fast kernel traps to implement thread primitives. Jeff Kimmel provided a number of other useful design suggestions. Many thanks to the following persons who contributed to the DG/UX implementation of POSIX Threads: Richard Barnette, Jeff Beneker, Gina Couch, Ming Hwang, David Lakey, Etta LeBlanc, Ed McAdams, Earle MacHardy, Bill McGrath, Mark O'Connell, Cyril Sagan, Lee Sanders, Ed Savage, Guy Simpson, Eric Vook. Special thanks to Tom Ash, Dean Herington, and Andy Huber for their careful review of this paper.

10. References

- [Alf94] R.A. Alfieri, "Pthread Groups and Hierarchical CPU Affinity," Data General Corporation, in progress.
- [And91a] T.E. Anderson, H.M. Levy, B.N. Bershad, and E.D. Lazowska, "The Interaction of Architecture and Operating System Design," *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 108-120, April 1991.
- [And91b] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism," *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pp. 95-109, October 1991.
- [Ber90] B.N. Bershad, T.E. Anderson, E.D. Lazowska, H.M. Levy, "Lightweight Remote Procedure Call," *ACM Transactions on Computer Systems*, volume 8, number 1, pp. 37-55, February 1990.
- [Dra91] R.P. Draves, B.N. Bershad, R.F. Rashid, R.W. Dean, "Using Continuations to Implement Thread Management and Communication in Operating Systems," *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pp. 122-136, October 1991.
- [Gol90] D. Golub, R. Dean, A. Florin, R. Rashid, "UNIX as an Application Program," *Proceedings of the Summer 1990 USENIX Conference*, pp. 87-95, June 1990.
- [Kel89] M.H. Kelley, "Multiprocessor Aspects of the DG/UX Kernel," *Proceedings of the Winter 1989 USENIX Conference*, pp. 85-99, January 1989.
- [Lie92] J. Liedtke, "Fast Thread Management and Communication Without Continuations," *Proceedings of Microkernel and Other Kernel Architectures*, pp. 213-221, April 1992.
- [Lie93] J. Liedtke, "Improving IPC by Kernel Design," *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pp. 175-188, December 1993.
- [Mar91] B.D. Marsh, T.J. LeBlanc, M.L. Scott, and E.P. Markatos, "First-Class User-Level Threads," *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pp. 110-121, October 1991.
- [Mot91] MC88110: Second Generation RISC Microprocessor User's Manual, Motorola Inc., 1991.
- [Mue93] F. Mueller, "A Library Implementation of POSIX Threads under UNIX," *Proceedings of the 1993 Winter USENIX Conference*, pp. 29-41, January 1993.
- [Ous90] J.K. Ousterhout, "Why Aren't Operating Systems Getting Faster as Fast as Hardware?" *Proceedings of the Summer 1990 USENIX Conference*, pp. 247-256, June 1990.
- [Pos93] POSIX P1003.4a/D8, *Threads Extension for Portable Operating Systems*, IEEE Computer Society, October 1993.
- [Pow91] M.L. Powell, S.R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks, "SunOS Multi-thread Architecture," *Proceedings of the 1991 Winter USENIX Conference*, pp. 65-80, January 1991.
- [Ste92] D. Stein, D. Shah, "Implementing Lightweight Threads," *Proceedings of the 1992 Summer USENIX Conference*, pp. 1-9, June 1992.
- [Wal92] J. Walpole, J. Inouye, R. Konuru, "Modularity and Interfaces in Micro-kernel Design and Implementation: A Case Study of Chorus on the HP PA-Risc," *Proceedings of the 1992 Workshop on Micro-kernel and Other Kernel Architectures*, pp. 71-82, April 1992.

Robert A. Alfieri works for Data General's UNIX kernel group in Research Triangle Park, NC, where he has developed threads, hierarchical affinity scheduling, virtual memory, and other kernel functionality. He received a B.S. degree in Computer Science from Northwestern University in 1985. His technical interests include operating systems, molecular nanotechnology, and Italian cooking. He can be reached at Data General Corporation, 62 T.W. Alexander Drive, Research Triangle Park, NC 27709, or electronically at alfieri@dg-rtp.dg.com.

Appendix A: Thread KFC Interfaces

This section lists all KFC interfaces for Pthreads and Pthread Groups. Like other UNIX systems [Pow91], DG/UX refers to kernel-level threads as LWPs (Light Weight Processes). However, DG/UX LWP interfaces differ and, as discussed earlier, DG/UX dedicates a very efficient LWP to each thread.

In all cases, a trivial “wrapper” function in the Pthread library invokes the appropriate KFC. Upon completion, the KFC returns values and status in the same way as a traditional system call. Because DG/UX ships the Pthread library only in shared form and does not publish the KFC interfaces, these interfaces can change from revision to revision without requiring that users relink their applications.

The following KFCs create, terminate, join, detach, and manipulate the scheduling of threads. In the case of `__lwp_create()`, the new thread begins execution at a pre-registered library function that invokes the thread’s start routine on the thread’s user stack.

```
int __lwp_create(
    lwp_sched_info_t    sched_info,
    lwp_stack_info_t    stack_info,
    void                (*start_rtn)(),
    void *
    lwp_group_id_t      group_id);

int __lwp_join(
    lwpid_t             lwpid);

int __lwp_exit(
    void *              exit_status);

int __lwp_detach(
    lwpid_t             lwpid);

int __lwp_set_sched(
    lwpid_t             lwpid,
    lwp_sched_info_t    sched_info);

int __lwp_get_sched(
    lwpid_t             lwpid);

int __lwp_yield(
    int                 to_peers,
    int                 is_global);

int __lwp_sleep(
    unsigned            seconds,
    unsigned            nseconds);
```

The following KFCs manipulate synchronization queues that are used to implement the contested cases for mutexes and condition variables. The Pthread library does not invoke KFCs in the uncontested cases.

```
int __lwp_sq_alloc(
    lwp_sq_info_t       creation_info);

int __lwp_sq_dealloc(
    lwp_sqid_t          sq_id);

int __lwp_sq_sleep(
    lwp_cond_t           cond_ptr,
    lwp_mutex_t           mutex_ptr,
    struct timespec *    abstime_ptr,
    lwp_sqid_t          sq_id);

int __lwp_sq_wakeup(
    lwp_sqid_t          sq_id,
    int                  is_broadcast);
```

The following KFCs create, destroy, and manipulate DG/UX Pthread Groups, which are discussed separately [Alf94].

```
int __lwp_group_create(
    lwp_sched_info_t    sched_info);

int __lwp_group_destroy(
    lwp_group_id_t      lwp_group_id);

int __lwp_group_set_sched(
    lwp_group_id_t      lwp_group_id,
    lwp_sched_info_t    sched_info);

int __lwp_group_get_sched(
    lwp_group_id_t      lwp_group_id);

int __lwp_group_get_times(
    lwp_group_id_t      lwp_group_id,
    struct timespec *   user_time_ptr,
    struct timespec *   sys_time_ptr);
```

UNIX is a registered trademark of Unix Systems Laboratories, Inc.

DG/UX is a trademark of Data General Corporation.

SunOS is a trademark of Sun Microsystems, Inc.

SPARC is a trademark of SPARC International, Inc.

Using OS Locking Services to Implement a DBMS: An Experience Report

Andrea H. Skarra
AT&T Bell Laboratories

Abstract

The paper describes a black-box analysis of the locking facilities in several UNIX-compatible operating systems for their ability to support transaction synchronization. It assesses the facilities for their adequacy, flexibility, and performance. Most of the operating systems in the study provide adequate support for a simple two-phase locking transaction system that does not require customized or priority-based scheduling of lock requests. The performance depends on a variety of factors: the average execution time for a lock request varies directly with the number of concurrent locks in the system and indirectly with the number of files locked for a given number of lock requests. The request time is smaller when the locked files are local to the requesting process instead of remote, and when a process locks a file's segments in order of adjacency rather than randomly. For the areas in which the OS provides inadequate support, the paper proposes several specific remedies.

1 Introduction

Applications typically interact with a database in the context of transactions to maintain consistency. A transaction is a sequence of operations that satisfies all consistency constraints on a database, and the database management system (DBMS) synchronizes concurrent transactions to produce a *serializable* execution (i.e., one that produces the same effect as some serial execution of the transactions) [3]. Thus, a database remains consistent across repeated and concurrent access when an application uses transactions.

Most commonly, the DBMS uses locking to synchronize transactions. The transactions request locks in a two-phase protocol that guarantees serializability, and a lock manager services the requests. Under the protocol, a transaction neither releases nor weakens its locks (e.g., from an exclusive to a share lock on the same object) until after it obtains all its locks.

The lock manager detects conflict and deadlock among the transactions, and it implements a scheduling algorithm for allocating the locks. Frequently, the lock manager is a server process: transactions send messages to the server to request locks, and the server maintains the data structures necessary for scheduling and for conflict and deadlock detection. Each such lock server is dedicated to a single DBMS.

We are developing a DBMS that uses the locking services of the operating system (OS) instead of a dedicated lock server [5]. The OS (i.e., a POSIX.1 compatible [6] together with NFS) supports fine-grained *Read* (Share) and *Write* (Exclusive) locking at the byte level for both local and remote files in a local area network (LAN) via the system call `fcntl()`. Each transaction requests locks from the OS in a two-phase protocol, and it commits in a way that guarantees atomicity. Interprocess communication between a client transaction and a database server is replaced by direct interaction with the OS to acquire both data items and *Read/Write* locks.

The advantages of using an OS locking facility to a DBMS developer include the following:

- *Ease of implementation*
An OS locking facility is an alternative to a dedicated lock server, and it makes the task of implementing one unnecessary for the developer.
- *Reduced system size and complexity*
The OS facility is already present; it does not add to system size as does a dedicated lock server.
- *Openness*
An OS locking facility is a service open to any application (i.e., the facility is not buried within a closed, monolithic system). It can be used for synchronizing across applications.
- *Rapid detection of process failure*
The kernel knows sooner than user-level pro-

cesses when to release locks due to process failure.

- *Potential for performance improvement*

For a given lock manager (i.e., data structures and algorithms), an application request for a lock is faster when the manager is implemented in the kernel rather than as a user-level process due to fewer context switches.

The paper assesses the OS locking services in terms of support for transaction synchronization. It considers the performance of the OS in granting lock requests, the ease of implementing a protocol that guarantees serializability, and the ability to customize the system for prioritized scheduling schemes. The assessment is a black-box analysis; it does not consider the actual OS code. Moreover, it covers only the OS locking services; the applicability of other OS capabilities to DBMSs is beyond the scope of the paper.

The contribution of this work is that it identifies requirements that OS locking services must satisfy to provide adequate and flexible support for transaction synchronization. It also identifies the areas in which the services provide inadequate support, and it proposes several specific remedies.

The paper first gives an overview of the OS locking services as defined by the documentation. It then describes the system's behavior in terms of lock allocation, deadlock detection, and scheduling, as determined empirically in our study, and it assesses the system for its ability to support a protocol that guarantees serializability. The paper then presents the results of a performance study, and it concludes with an overall assessment of the services together with some recommendations.

2 OS Locking Services

Several commercial OSs provide a locking facility through which processes can lock (parts of) files on the same host. We consider the following OSs: HP-UX 9.01 (UX), IRIX 5.1.1.2 (IRIX), RISC/OS 4.52B (RISC), Solaris 2.1 (Solaris1), Solaris 2.3 (Solaris3), SunOS 4.1.1 (SunOS), and UNIX System V Release 4 (UNIX). The OSs support the same locking interface and compatible protocols, and their file systems can be integrated in a LAN under the Network File System (NFS) for a distributed configuration. NFS transparently propagates lock requests to remote hosts [2, 11].

For the remainder of the paper, the term OS refers generically to any of the systems. We distinguish OSs only when there is a discernible difference in their behavior or performance.

2.1 Locking

The OS supports *Read* and *Write* lock modes with the usual conflict semantics: a *Write* lock conflicts with a *Read* or *Write* lock held by a different process on the same file segment; *Read* locks do not conflict with each other.

A process requests a lock on an object with or without blocking. If it issues a blocking lock request that conflicts with some other lock on the object, the OS suspends the process and queues the request until the process that holds the conflicting lock releases it. If the request is nonblocking instead, or if queuing would result in deadlock, the request returns immediately with an error code.

The lockable entities in the OS are files. A process can lock all or part of a file; the smallest lockable unit is a byte. A process can lock nonexistent bytes that are partially or totally beyond the physical end-of-file (EOF). It is also possible to lock just the EOF itself to prevent creation of new slots in the file.

Each lock is associated with a file and a process. When a process terminates, or when it closes a file, the OS releases any locks the process holds on the file automatically. A process can explicitly release locks as well. Locks are not inherited by child processes, and they are not transferable to another process.

The kernel manages the locking of local files, and it may impose a limit on the number of lock requests that it manages at a time. For locking remote files, a user-level daemon (*lockd*) runs on each machine in the LAN. The *lockd* on a specific host *H* handles lock requests from the *H* kernel for files on other hosts, and it handles lock requests from *lockds* on other hosts for files on *H*.

To illustrate, suppose a process on host *H*₁ requests a lock on a file *f*. If *f* is local to *H*₁, the kernel alone handles the lock request. If instead *f* is on a different host *H*₂, the kernel passes the request to its local *lockd*, which passes it to the *lockd* on *H*₂, which passes it to the *H*₂ kernel. The kernel grants the request if no other process holds a conflicting lock on *f*; otherwise, it denies or queues the request. The result returns to the *H*₁ process via the same path.

If a remote server crashes, the lock daemon tries to restore locks that were held by processes. If it cannot reinstate a particular lock, it sends a SIGLOST signal to the process, and the process must abort. The OS does not provide a function through which a process can find out which locks it currently holds.

The OS supports both *mandatory* and *advisory* locks, although POSIX.1 includes only the latter. Mandatory locks block file reads and writes until conflicting locks are removed, while advisory locks are used in protocols that processes voluntarily follow. The study covers only advisory locks, being safer and sufficient for a DBMS. A runaway process that fails to release a mandatory lock can hang or crash the system, and unauthorized access can be prevented via mechanisms other than locking.

2.2 Syntax

The signature of the system call that a process uses to request a lock on (a part of) a file is the following:

```
int fcntl(fd, cmd, arg)
int fd, cmd, arg;
```

where *fd* is a file descriptor returned by *open()*, *arg* is a pointer to a structure that specifies the location and number of bytes to be locked and the desired lock mode (i.e., *F_RDLCK* (*Read*), *F_WRLCK* (*Write*), or *F_UNLCK* (*UnLock*)), and *cmd* is one of the following:

F_SETLK Set or clear the file segment lock *l* described in *arg*. If another process holds a lock that conflicts with *l*, fail and return immediately. The requesting process must retry to get *l*.

F_SETLKW Same as **F_SETLK**, except the process blocks until no other process holds a conflicting lock and *l* is granted. If blocking the process would result in deadlock, fail and return immediately.

F_GETLK Get a description of the first lock that conflicts with *l*. If such a lock exists, return that lock's description, including the identifier of the process that holds the lock. Otherwise, return *F_UNLCK*, whether the requesting process already holds *l* or not.

The OS recognizes special settings for the fields in the *arg* structure that allow a process to lock the entire file or to lock the EOF and beyond with one *fcntl()* invocation.

3 Deadlock detection

When a process issues a blocking lock request that conflicts with some current lock on the object, and queueing the request would result in deadlock, the request fails and returns immediately with an

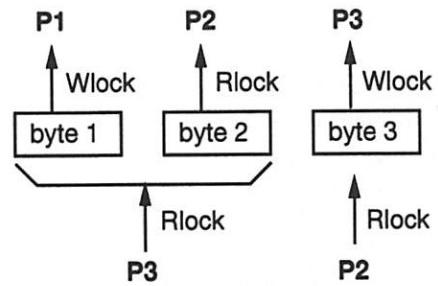


Figure 1: A deadlock detection example

error code. The process then decides on an action: it can retry after some delay, it can request a different lock, or it can abort.

We found that the OSs (with NFS) detect deadlock among processes on the same or different machines in a LAN, provided the files they access are on a single host, and they correctly handle subtle cases such as the following. In Figure 1, processes *P*₁ and *P*₃ hold *Write* locks on bytes 1 and 3 respectively; *P*₂ holds a *Read* lock on byte 2. *P*₃ requests a *Read* lock on bytes 1 and 2, and *P*₂ requests a *Read* lock on byte 3. Although *P*₂ is waiting for *P*₃ at byte 3, and *P*₃ is waiting for *P*₂ at byte 2, deadlock is not actually present. *P*₃ is really waiting for *P*₁ at byte 1. Accordingly, the OS suspends both *P*₂ and *P*₃, and does not refuse either request due to deadlock. When *P*₁ terminates, *P*₃ gets its locks and continues processing, and when *P*₃ terminates, *P*₂ gets its lock.

Importantly, however, we found that the OSs (with NFS) do *not* detect deadlock across files on different machines, regardless of whether the processes are on the same or different machines. An application's data must reside on the same machine, or the data must be partitioned such that no transaction accesses data on more than one machine. Otherwise, processes may hang.

For example, suppose *H*₁ and *H*₂ are hosts in a LAN, and *P*₁ and *P*₂ are processes that run on *H*₁ but access files on both *H*₁ and *H*₂. If *P*₁ *Write*-locks a file on *H*₁, and *P*₂ *Write*-locks a file on *H*₂, then both processes hang if each then requests a lock on the file locked by the other process. Deadlock across files on different hosts goes undetected, regardless of where the processes execute: the processes still block if *P*₂ runs on *H*₂ or if both processes run on another host *H*₃.

4 Lock request scheduling

A *scheduler* is an algorithm that manages the lock request wait-queue for each object. It establishes the priority of lock requests in the queue with respect to each other and with respect to new lock requests. That is, it decides which of any queued requests to grant when a process releases locks, and it decides whether a new lock request conflicts with any queued requests.

We found that the OS uses a *FIFO/Reader's Priority* scheduling algorithm. It processes an object's wait-queue in first-in-first-out order to find lock requests that it can grant when a process releases a lock on the object. If a *Read* lock is the next grantable request in the queue, however, the algorithm grants *all* the queued *Read* requests, even those that follow *Write* requests in the queue.

We also found that the OS does not consider an object's wait-queue when it tests a new lock request for conflict; it considers only the current locks on the object. Consequently, new *Read* requests also have priority over queued *Write* requests. A *Read* request on a *Read-locked* object is always granted, even if an earlier *Write* lock request is still in the queue.

4.1 Fairness

In any *Reader's priority* scheme, the possibility of starvation exists for transactions that request *Write* locks. Moreover, we found that the OS does not grant any part of a lock request until it can grant the lock on the entire file segment. If a process P requests a *Write* lock on bytes 0-1023 of a file f , the OS does not grant locks to P for any part of the segment if some other process P' holds a *Read* or *Write* lock on a byte $i \in \{0..1023\}$. Moreover, the OS does not give P any priority at the bytes where there is no conflict at the time of the request. If another process P'' requests a *Write* lock on another byte $j \in \{0..1023\}$ before P gets its lock, the OS grants the P'' request, and P must wait for P'' to terminate as well as P' . P could wait forever.

The DBMS application developer must work around the scheduling algorithm and its fairness characteristics to get better performance (albeit suboptimal), since the scheduler itself cannot be changed. The developer might program P to request its lock on the file segment one byte at a time, for example. The unhappy choice is between extra system calls in P or possible starvation.

4.2 Upgrading locks

A process may request a *Write* lock on an object that it already has *Read-locked*. We found that all the OSs but Solaris1 handle lock upgrades in a way that supports two-phase locking: the scheduler places the *Write* request at the head of the wait-queue and grants it as soon as there are no other *Readers*. It replaces the *Read* lock with the *Write* lock before it grants any other (queued) *Write* requests. If more than one *Reader* requests an upgrade, the OS returns a deadlock error to all but the first request.

In contrast, Solaris1 handles the same request as follows: it releases the *Read* lock and places the *Write* request at the end of the wait-queue. Any queued *Write* requests are granted before the upgrade. If two *Readers* request an upgrade, Solaris1 grants each in turn; it does not report deadlock.

An upgrading process cannot tell whether another process got a *Write* lock during the upgrade. Thus, it cannot tell (without rereading the data) whether the data it originally read with the *Read* lock is still valid when it obtains the *Write* lock.

Given the Solaris1 scheduler, it is impossible to implement a synchronization protocol that guarantees serializable executions, unless *Read* locks are used only when there is zero chance for update. This strategy substantially reduces potential concurrency.

Parenthetically, the Solaris1 policy is just as inappropriate for any other OS application that is using the locks to maintain a cache.

4.3 Capacity

We found that several OSs limit the number of lock requests that they manage at a time for local files (Table 1).¹ The other OS capacities are virtually unlimited: they handled up to 81,920 lock requests by concurrent processes on multiple files in our measurements.

When an OS reaches its limit, `fcntl()` returns a special error code (`ENOLCK`) rather than granting or queueing new lock requests. The OS accepts new requests again when locks are later released.

Solaris1, however, malfunctions when the number of lock requests exceeds its capacity. Up to the limit, the OS grants locks and detects conflict as it should. Beyond that, however, `fcntl()` returns without error to all lock requests except for those on bytes that were locked before the OS reached

¹In UNIX, the file system parameter `FLCKREC` defines the capacity. It can be reset to a value between 100 and 2000 inclusive upon recompiling the kernel [1].

OS	Machine	Maximum locks
RISC	MIPS Magnum	100
UX	HP 9000/730	195
UNIX	AT&T Starserver E 486	291
Solaris1	Sun SPARC LX	508
Solaris3	Sun SPARC 1	510

Table 1: Experimentally determined capacity of several locking facilities

its limit; to these it returns ENOLCK. As a result, Solaris1 grants (i.e., does not delay or deny) conflicting lock requests for any bytes that were not locked before it reached its limit. Obviously this is a major problem.

We found that the capacity of an OS comprises the combined total of locks currently granted and requests that are queued, and it is independent of the number of processes and files involved. Moreover, the limit applies to the files local to the OS: a process that executes on host H_1 and locks files on H_2 is subject to the capacity of H_2 rather than H_1 . Finally, the limit applies primarily to locks on nonadjacent file segments. In Solaris3, a process can lock any number of file segments that are adjacent to a previously-locked segment, provided that segment is the one among all the locked segments with either the highest or lowest file offset.

The capacity of an OS affects its scheduling behavior. In particular, the scheduling policy becomes *retry* when an OS reaches its limit. A process whose lock request returns with ENOLCK must resubmit the request in order to obtain the lock; the scheduler does not absorb the request in a wait-queue and apply it later.

Importantly, processes can become *live-locked* under a *retry* policy. If two processes each request a lock that the other process holds without releasing any that the other process needs, then they will retry forever. If they quit and restart, they may generate the same live-lock situation again. In either case, the potential exists for their making no forward progress.

4.4 Priority

A priority-based scheduling algorithm is one that recognizes an application-defined ordering over a set of transactions. It uses the ordering to

prioritize the wait-queue and to revoke locks. Priority scheduling is important in real-time DBMSs [8] or in any application where the processes have unequal significance.

The OS supports priority-based scheduling, but only in a very limited sense. A process whose lock request is denied due to conflict can discover the identifiers of processes with conflicting locks; it can then send signals to kill lower priority processes. Only in a single-machine environment is the mechanism even possible, however, since process identifiers contain no information about the host machines. Moreover, processes waiting in the queue cannot be identified or reordered; it is only possible to preempt lock holders.

The ability to reorder the wait-queue would be very useful in deadlock resolution. Currently, the OS resolves deadlock by refusing the latest lock request, a reasonable policy when the detection algorithm runs every time a lock request is about to be queued and all processes have equal priority and capabilities. When equality is not the case, however, the optimal resolution strategy is likely to involve dequeuing a specific process, rather than the one that happens to be last.

5 Lockable entities

The OS facility supports locking at arbitrarily fine granularity, and as a result, it can provide support for transaction synchronization. Earlier studies criticized prior OS locking schemes for their coarse granularity [10]. In addition, the facility integrates locking at the level of records, pages, and files under a uniform interface.

The facility has a problem, however, in that it associates locks with open file descriptors. A process must open a file to lock it, and if it closes the file, the OS automatically releases any locks that the process holds on the file. Typically the number of open file descriptors per process is limited, and a transaction may have to access and lock a larger number of files.

The hard limit on open file descriptors varies in the systems we tested from 256 in SunOS to 2500 in IRIX, with Solaris3, HP-UX, and UNIX having a limit of 1024. In most database applications, a process is unlikely to lock more than 2500 files. Some applications, however, will easily exceed 256 open file descriptors. For example, a process that creates a transaction-consistent copy of the database as a backup must open and lock all the database files. In a DBMS that supports *horizontal partitioning* (i.e., separating a relation's

records into different files according to their attribute values), the number of files could be large. One application within our company partitions its data into over 800 files.

When it is possible for some transaction to exceed the limit, the synchronization protocol must resort to using lock files or a listing file l that contains an entry for every file in the database. A transaction T that holds a lock on file f must create a lock file for f or lock the entry for f in l before it closes f . To do so, T must first obtain a lock on all of f , even if it has only one byte locked. Moreover, for every file f that T locks, T must test for the existence of a lock file or a lock on listing before it requests its first lock on f . Both options substantially reduce concurrency, and they increase complexity and overhead with extra file maintenance.

6 Downgrading locks

None of the OSs provide a way to prevent a transaction from inadvertently downgrading locks and violating serializability. If a transaction locks a record in *Write* mode, and later requests a *Read* lock on the same record, the OS grants the weaker lock and releases the stronger one.

The situation most commonly arises when a transaction queries several indices and obtains the same record multiple times. For example, a transaction T queries a relation R via the *color* index, and the query returns a record r . T gets a *Write* lock on r and changes its color. T then queries R via the *size* index, and it again receives r . T does not change r in this case, so it only gets a *Read* lock on r , inadvertently downgrading the *Write* lock it already has.

A process cannot find out from the OS what locks it holds (e.g., before a request for a *Read* lock) to avoid downgrading *Write* locks. Even if it could, however, the extra system calls that the approach requires reduce its viability. Transactions must either request locks in a way that avoids the risk of downgrading, or they must remember which file segments they have *Write*-locked. Neither approach is entirely satisfactory. Transactions can avoid downgrading if they delay all *Write* lock requests until after all *Read*-locking is complete (e.g., just before commit), but they run a higher risk of deadlock. If they keep track of their *Write* locks, they incur more overhead, and they duplicate information that the OS already stores.

The downgrading problem can be solved easily if the OS simply adds a new lock mode, *Read'*. A

request by transaction T for a *Read'* lock on an object o is the same as a request for a *Read* lock on o , except that if T already holds a *Write* lock on o , the *Read'* request does not change it.

7 Performance

We completed several performance studies that measure the effect of the following variables on the time required for a process to obtain locks on a file:

Lock type: getting *Read* vs. *Write* locks

Fragmentation: locking contiguous vs. noncontiguous bytes in a file

Access order: locking noncontiguous bytes in ascending order of file location vs. descending order vs. random order

File location: locking local vs. remote files

Number of files: getting n locks on one file vs. ten files ($n/10$ locks on each)

Concurrency: one vs. ten concurrent processes each getting n locks on the same file

We performed the studies on the following OSs:

SunOS: SunOS 4.1.1 on a Sun SPARC 1+

Solaris3: Solaris 2.3 on a Sun SPARC 1

IRIX: IRIX 5.1.1.2 on a SGI Indigo

The *reference* for each measurement is a single process that requests *Read* locks from the OS on noncontiguous bytes in a single local file (i.e., every second byte in ascending order of file location). The test process differs from the reference process in each case only in the variable being studied.

We also ran the reference process on the following systems to compare the OS locking services to a dedicated lock server that executes in user space:

SunOS2: SunOS 4.1.1 on a Sun SPARC 2

EXODUS: Exodus storage manager server 2.2 on SunOS 4.1.1 and a Sun SPARC 2

Comparisons that involve remote locking were conducted overnight, and for the local locking experiments, the machine was not shared. In both cases, the network load was minimal. We used the *time()* command to measure the system and real times of process execution.

We describe the experimental results in the next section. Because the study is a black-box analysis, the interpretation of the results is not as specific as an interpretation based on the actual OS code would be. Throughout the discussion, however, we speculate on the nature of the data structure that the OS uses to represent the locks (i.e., the *locking structure*). The locking structure for a file must represent the ranges of bytes that are locked, by

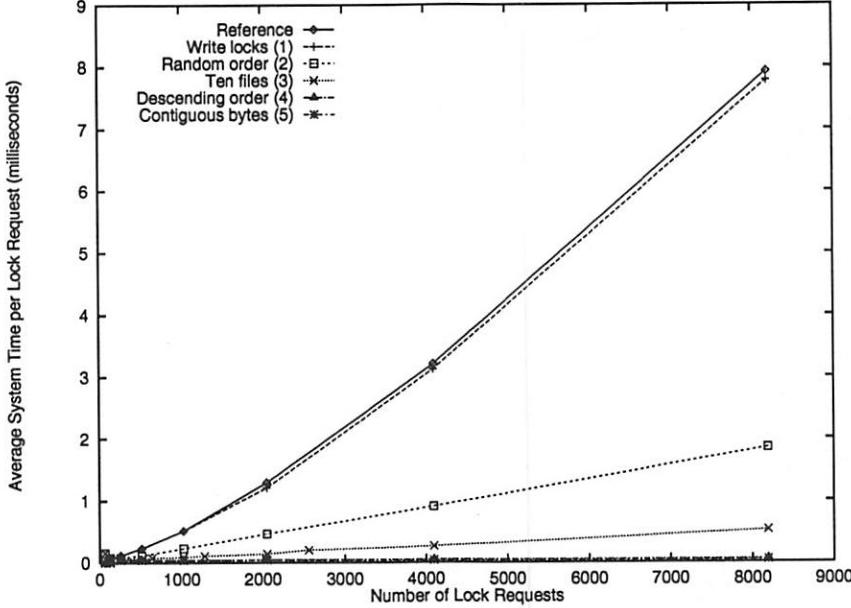


Figure 2: IRIX comparison. The average system time per lock request of the reference process is compared to that of the following (%age of the reference time at 8192 lock requests): (1) a process that gets *Write* locks (no difference), (3) a process that locks ten files (one-tenth of the locks on each) (6.6%), (5) a process that locks contiguous bytes (0.6%), and (2) one that locks noncontiguous bytes in descending offset order (0.5%), and (4) one that locks noncontiguous bytes in random order (23.4%).

which process(es), and in what mode. When the OS receives a lock request for a segment in file f , it must search f 's locking structure to detect conflict with any locks held by other processes on overlapping segments. If there is no conflict, it enters the lock information in the locking structure and returns. Otherwise, it runs the deadlock detection algorithm and either queues or denies the request.

7.1 Results

The most marked and consistent results are the following: locking contiguous bytes is faster than locking noncontiguous bytes, locking local files is faster than locking remote files, obtaining n locks on one file is slower than obtaining n locks on ten files ($n/10$ locks on each), and the time per lock request is smaller when one process obtains n locks on a file than when ten concurrent processes each obtain n locks on the same file. The effect of the other two variables, lock type and access order, differs among the OSs.

To compare processes that are on the same machine as the files they lock, we calculate the *average lock request system time* $(ST_n - ST_0)/n$, where ST_n is the system time for a process to obtain n locks, and for the other comparisons, we cal-

culate the *average lock request real (elapsed) time* $(RT_n - RT_0)/n$, where RT_n is the real time for a process to obtain n locks.

We first discuss the variables that reduce the lock request time or have no effect, and then we consider concurrency and remote locking. We accompany the textual description of the results with several figures. For each figure, we encourage the reader to first locate the data for the reference process (perhaps marking it with a colored pen) and then consider the other data in relation to the reference.

Fragmentation

We compare the average lock request system times of a reference process R to those of another process P_C that executes on the same machine at a separate time. P_C gets *Read* locks on *contiguous* bytes in a single local file (i.e., every byte in order of file location, such that the region it locks is not fragmented), whereas R locks noncontiguous bytes. The results are the same whether P_C requests the locks in ascending or descending order (Figures 2-4).

In the capacity measurements for Solaris3 (Section 4.3), we observed that a process that requests locks on contiguous bytes in either ascending or

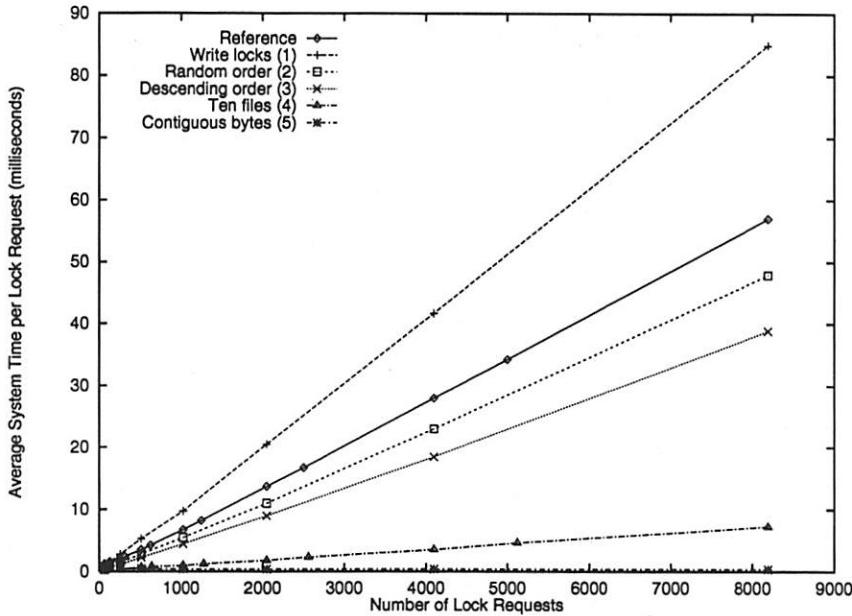


Figure 3: SunOS comparison. The average system time per lock request of the reference process is compared to that of the following (%age of the reference time at 8192 lock requests): (1) a process that gets *Write locks* (149%), (4) a process that locks ten files (one-tenth of the locks on each) (12.7%), (5) a process that locks contiguous bytes (0.6%), (3) one that locks noncontiguous bytes in descending offset order (68.1%), and (2) one that locks noncontiguous bytes in random order (84.0%).

descending order can obtain an unlimited number of locks. When the bytes are not contiguous, however, the process obtains no more than 510 locks. This suggests that the locking structure entry for a locked file segment contains fields for *first-byte* and *last-byte* that are decremented or incremented when the process locks an adjacent segment. Indeed, a process that initially obtains 509 locks on a range of noncontiguous bytes can subsequently obtain an unlimited number of locks on bytes that are adjacent to either end of the range. If the process instead requests locks on some (unlocked) bytes in the midst of the range, however, it can obtain only one more lock. This suggests that the OS modifies the *first-* and *last-byte* fields only in the locking structure's endpoint entries. It does not modify or merge entries in the middle.

A locking structure such as this explains the marked improvement in P_C 's lock request timings as compared to R 's. In particular, a lock request on a file f by P_C is simply two context switches (user \rightarrow kernel and kernel \rightarrow user) and an increment or decrement operation. Indeed, P_C 's lock request timings represent a reasonable upper bound for the context-switching overhead in `fcntl()`, since the computation on the locking

structure is minimal.

Access order

We compare the average lock request system times of a reference process R to those of processes P_D and P_R which execute on the same machine as R at separate times. Like R , P_D and P_R get *Read* locks on noncontiguous bytes in a single local file (i.e., every second byte), but R locks bytes in ascending order of file location, while P_D and P_R lock bytes in *descending* and *random* order respectively. We generated randomly ordered, noncontiguous bytes by rounding each number generated by `rand()` to an even integer.

In both IRIX and SunOS, P_D and P_R are much faster than R (Figures 2,3), although the magnitude of the difference is greater in IRIX: P_D 's times in IRIX are comparable to those of a process that locks contiguous bytes. In Solaris3, however, the effect of access order on the lock request time is not significant for the number of lock requests that are within the OS capacity (Figure 4).

The results are consistent with a locking structure whose entries are ordered by file location, and an access algorithm that does a search and insertion for each lock request; the search locates the

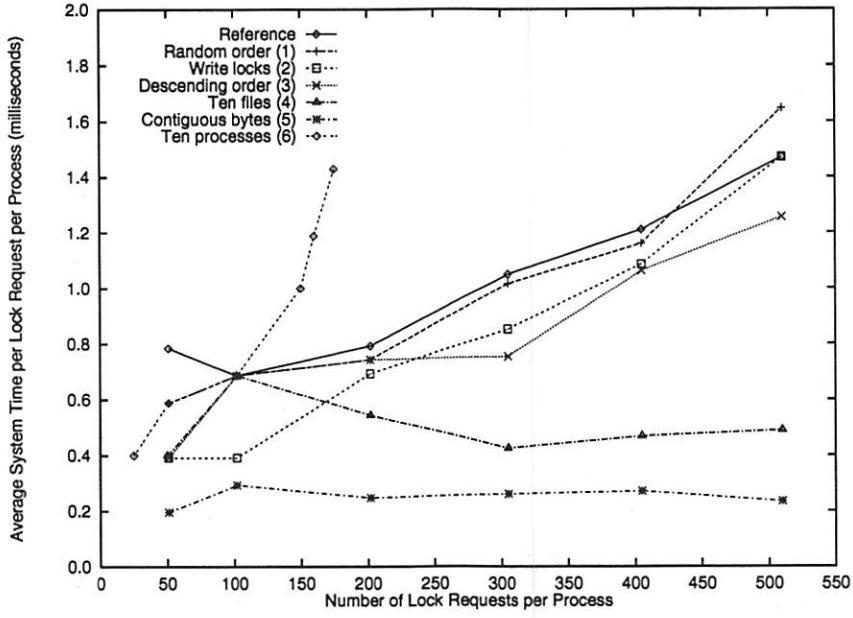


Figure 4: Solaris3 comparison. The average system time per lock request of the reference process is not markedly different from that of a process that gets (1) randomly ordered locks, (2) *Write* locks, or (3) locks in descending offset order, but it is greater than that of (4) a process that locks ten files (one-tenth of the locks on each) (44.8% of the reference time at 510 requests) or (5) a process that locks contiguous bytes (26.5% of the reference time at 510 requests). When ten concurrent instances of the reference process request locks on the same file (6), the average request time is 190% of the single reference time at 175 locks/process (with the largest standard deviation being 0.85 at 25 locks). The processes could obtain no more than 175 locks apiece without exceeding the OS capacity.

point in the locking structure where the entry for the lock is inserted. Given such a locking structure, P_D and P_R are both faster than R due to a shorter search path.

Lock type

We compare the average lock request system times of a reference process R to those of another process P_W that executes on the same machine at a separate time. P_W gets *Write* locks on noncontiguous bytes in a single local file (i.e., every second byte in ascending order of file location), whereas R gets *Read* locks.

R is faster than P_W in SunOS, but there is not a marked difference between them in either IRIX or Solaris3 (Figures 2-4). It is unclear why *Write* locks should be more costly in SunOS.

Number of files

We compare the average lock request system times of a reference process R to those of another process $P_{F_{10}}$ that executes on the same machine at a separate time. Like R , $P_{F_{10}}$ gets *Read* locks

on noncontiguous bytes (i.e., every second byte in ascending order of file location), but R requests n locks on a single local file, while $P_{F_{10}}$ requests $n/10$ locks on each of ten local files.

$P_{F_{10}}$ is faster than R on IRIX, SunOS, and Solaris3 (Figures 2-4). It is likely that the OS maintains separate locking structures for different files, since locks on different files do not conflict. Maintaining separate locking structures yields a performance advantage by reducing the search path for each lock request.

The improvement is not linear in the number of files, however. In IRIX, the average time at 8192 lock requests for $P_{F_{10}}$ is 6.6% of R , whereas the average time for a process $P_{F_{20}}$ that divides the locks among 20 files is 4.2% of R . In SunOS, $P_{F_{10}}$ and $P_{F_{20}}$ are 12.7% and 8.2% of R respectively.

The average lock request time increases with the current total number of lock requests granted and managed by the OS. In IRIX, for example, the average request time for $P_{F_{10}}$ is 9.57 milliseconds when the system total is 81920 locks and only 7.92 milliseconds when it is 8192.

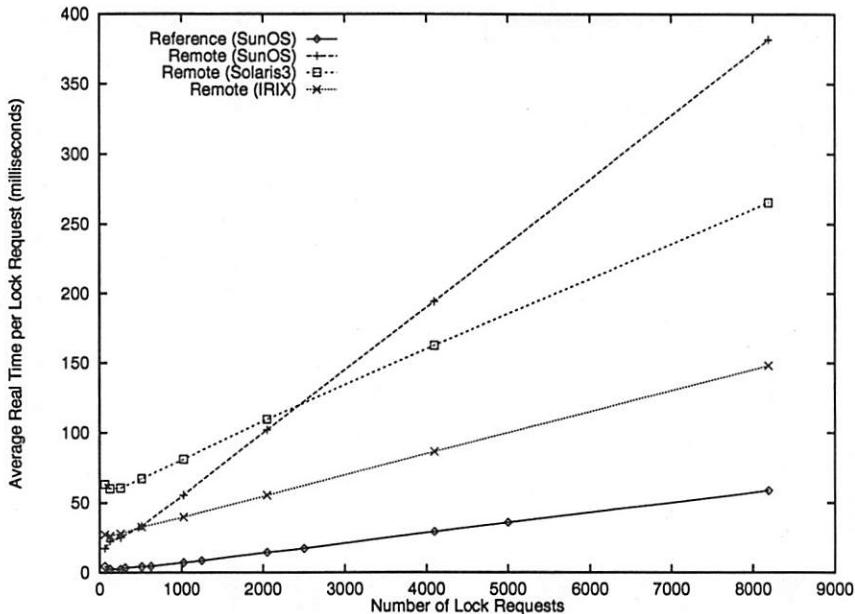


Figure 5: Locking a remote SunOS file from IRIX, Solaris3, and SunOS. The average real time per lock request of a reference process that locks a local file under SunOS is less than that of a remote process that locks the same file. The average time at 8192 lock requests as a percentage of the local process time is 652% for a remote SunOS process, 454% for a Solaris3 process, and 253% for an IRIX process.

The same relationship holds in SunOS, although we could not measure the average request time for $P_{F_{10}}$ at a system total of 81920 locks. On each of three attempts to get the value, the workstation crashed. The lock-requesting process ran for approximately 80 minutes real time and acquired between 76000 and 77000 locks.

Clearly, the average lock request time is affected not only by the search time but also by the insertion time. As the locking structures grow and use more space, it is reasonable to expect that the memory management will become more time-consuming.

File location

We compare the average lock request real times of a reference process R that locks a local file f under SunOS to those of processes P_I , P_{S3} , and P_S . Like R , they get *Read* locks on noncontiguous bytes in f (i.e., every second byte in ascending order of file location), but they execute remotely at separate times under IRIX, Solaris3, and SunOS respectively.

R executes much more quickly than do the remote processes (Figure 5). Their average times at 8192 lock requests as a percentage of R 's time is 253% for P_I , 454% for P_{S3} , and 652% for P_S .

The results are explained in part by the additional context switches and interprocess communication associated with the NFS remote locking implementation (i.e., *lockd* Section 2.1).

To approximate this overhead, we ran P_{C_r} on each of the remote OSs, a process that *Read*-locks contiguous bytes in f . Recall that lock requests for contiguous bytes require so little computation that their execution times approximate context switching alone (Figures 2-4). At 8192 lock requests, the average times for P_{C_r} on IRIX, Solaris3, and SunOS are 19.9% of P_I 's time, 22.6% of P_{S3} 's time, and 7.89% of P_S 's time respectively.

The overhead approximation, however, does not entirely account for the request time differences between R and the remote processes. In each OS, the average real time for the remote process (i.e., P_I , P_{S3} , or P_S) is greater the sum of the overhead (i.e., P_{C_r}) and the local locking time (i.e., R). More experimentation is needed to account for the difference.

Concurrency

We compare the average lock request system times of a single reference process R to those of ten concurrent instances of the reference process that execute on the same machine as R at a separate

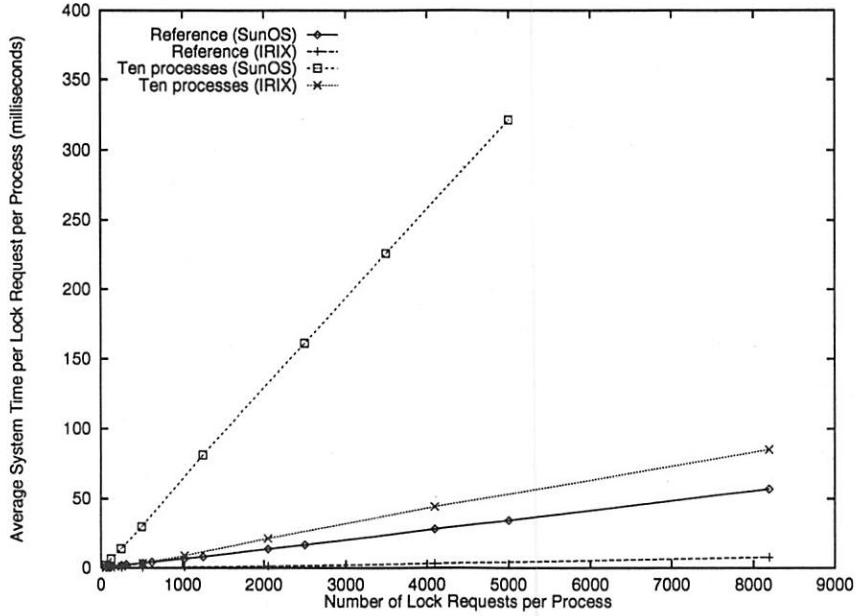


Figure 6: Concurrency in IRIX and SunOS. The average system time per lock request per process of a single reference process is less than that of ten concurrent instances of the reference process that request locks on the same file. The average time of the concurrent processes in IRIX is 1080% of the single reference time at 8192 locks/process (with the largest standard deviation being 1.04 at 4096 locks) and 939% in SunOS at 5000 locks/process (with the largest standard deviation being 4.56 at 500 locks).

time and obtain locks on the same file.

R executes much more quickly than do the concurrent processes (Figure 6). In IRIX, the average time at 8192 lock requests/process is ten-times that of *R*, and in SunOS, the average time at 5000 lock requests/process is nine-times that of *R*. Consequently, the real (elapsed) time for the ten processes is approximately 100-times that of the single process. We could not measure the effect of concurrency in Solaris3 to the same extent due to its limited locking capacity, but the results suggest the same trend (Figure 4).

When the ten concurrent processes each lock a different file rather than the same file, the average request times are close to those of a single process that executes alone. The OS handles the same number of lock requests, but they are spread across more files, and hence the locking structure for each file is smaller. At 8192 lock requests/process, the average request time is 103% of *R*'s time in IRIX and 101% of *R*'s time in SunOS.

We also compared the average lock request system times of *R* to those of ten concurrent processes, where nine processes request *Read* locks on the even-numbered bytes in a file, and one requests *Write* locks on the odd-numbered bytes. The dif-

ference in request times between the ten processes with all *Readers* and the ten processes with one *Writer* was negligible, although in SunOS, the process getting the *Write* locks consistently finished first.

Dedicated lock server

We compare the average lock request real times of the reference process *R* to those of another process *P_E* that executes on the same machine at a separate time. *P_E* obtains *Read* locks from a dedicated lock server in EXODUS, a DBMS that executes in user space [4], whereas *R* obtains *Read* locks from SunOS2.

EXODUS provides automatic page-level locking for objects that a transaction reads or modifies. To isolate the cost associated with just the locking mechanism, we used an internal function that acquires a file lock from EXODUS without reading in any objects.²

P_E gets locks faster from EXODUS than *R* gets them from SunOS2 (Figure 7). *R* requests locks on the same file, however, while *P_E* requests locks on different files. Consequently, the lock alloca-

²The (undocumented) function, `rpc.LockFile()`, was suggested by Mike Zwilling at the University of Wisconsin.

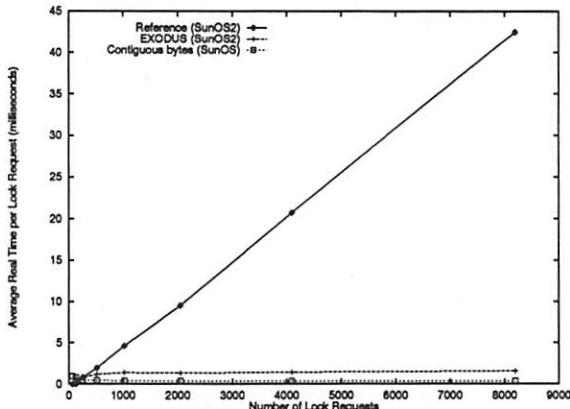


Figure 7: A dedicated lock server on SunOS2. The average real time per lock request of a reference process on SunOS2 is greater than that of a process that obtains *Read* locks from EXODUS, but the EXODUS process is not as fast as a SunOS process that locks contiguous bytes. The average time at 8192 lock requests as a percentage of the reference time is 3.7% for the EXODUS process and 0.8% for the other.

tion algorithm in SunOS2 iterates over a locking structure that grows with each new lock request, whereas EXODUS simply adds an entry to the locking structure: P_E locks a different file with each request.

A fairer comparison would have included a SunOS2 process that gets one lock on each of n different files. Due to the limitation on the number of open file descriptors, however, we could not gather a sufficient amount of data. So instead we include a SunOS process that locks contiguous bytes. Like P_E , its requests do not require a locking structure search. This process obtains locks faster than P_E .

7.2 Assessment

The performance profile of the OS locking service must be improved, especially with regard to sequential access, concurrent processing, and remote file locking, if it is to provide adequate support for transaction synchronization in a DBMS.

Specifically, the lock allocation algorithm should handle sequential as well as random access to the records in a file gracefully. Transactions frequently use sequential access to get better paging dynamics and to reduce deadlock (i.e., a common strategy for deadlock avoidance is to define an access order for data, and for the data within a file, the usual ordering is by location).

Further, a DBMS typically supports many concurrent processes that access the same data. One expects the elapsed time for n concurrent processes to be less than (or at least not much greater than) their sequential execution time (i.e., n -times the average single process time). A performance profile in which the elapsed time is n^2 -times the single process time is unacceptable for transaction synchronization.

Finally, the remote file locking service under NFS is very slow, and given the absence of deadlock detection across files on different servers, its usefulness is extremely limited.

8 Amenities

While using the OS locking facility, we identified several additional capabilities that would improve performance and convenience of use: timeouts, the ability to request a set of locks, and a transfer function.

Timeouts

A process advances in the wait-queue for a lock only while it blocks. For critical processes, however, we usually want to bound the duration of blocking with a timeout. Indeed, timeouts are required in a distributed setting where processes access both local and remote files, because the OS (with NFS) does not detect deadlock across file systems. A developer can program a timeout using `alarm()`, but as a convenience, the OS should provide a timeout parameter for `fcntl()` as it does for `select()`.

Requesting a set of locks

A single lock request covers a single contiguous region in a single file. For efficiency, the OS should provide a function `vfcntl()` that gets locks on a *vector (set) of regions* in the same or different files. When invoked with blocking, `vfcntl()` returns when it gets all the locks. Without blocking, it returns immediately and indicates which locks it got (a la `select()`).

In addition, `vfcntl()` could provide an alternative fairness model for scheduling. In particular, a process that locks a region r using `vfcntl()` with r defined as a set of bytes (i.e., `vfcntl({bytes_in_r}, ...)`) gets the lock on each byte as soon as it is available.

Transfer function

We strongly suggest a function through which one process transfers (a subset of) its locks to another. The transfer function is applicable to fault

tolerance (e.g., a sick process transfers its locks to a replacement process before exiting), scheduling (e.g., instead of blocking for a lock request, a process spawns a child, and the child blocks; when it gets the lock, the child transfers it to the parent process), and nested transactions (e.g., parent and child processes transfer locks according to a nested transaction protocol) [7].

9 Discussion

The OS provides a locking facility with which a DBMS developer can implement a synchronization protocol based on *Read/Write* locks. It is currently useful for applications whose transactions have the properties shown in Table 2.

While the OS facility has several advantages, such as openness, decreasing the time to implement a DBMS, and reducing the resulting system size, it also has several deficiencies that limit its applicability.

9.1 Requirements

The following are required areas for improvement:

- *Two-phase locking*

The OS must provide better support for two-phase locking, the most common protocol for producing serializable executions. The facility must handle both the upgrading and downgrading of locks as well as the closing of files without compromising correctness.

The Solaris1 protocol for upgrades must not be used in any OS, since it is inappropriate for both transaction synchronization and cache coherence algorithms. The downgrading problem is handled simply by adding the new lock mode *Read'*.

- *Performance*

The performance profile must be improved to better support sequential access, concurrent processing, and remote file locking.

- *Locking capacity*

The OS locking capacity must be increased in those systems that limit it. Even the largest of the limited capacities, a maximum of 510 in Solaris3, is too small for a DBMS. The scheduling policy in these cases degrades too soon to timeout and *retry* with the associated risk of live-lock.

An estimate for the number of locks required for the database benchmark TPC/A is 1800

on a DECstation 5000 class machine (At a 30 second user response time and 10 transactions per second) [9].

- *Deadlock detection*

The OS should detect deadlock across files on different machines to support location independence of data and to reduce the need for timeout and *retry* strategies.

- *Documentation*

The locking facility is inadequately documented. A developer cannot determine from the documentation whether a synchronization protocol based on the facility guarantees serializability. Moreover, the documentation is incorrect in some cases. SunOS, for example, describes its upgrade protocol to be the one we found in Solaris1. In reality, however, SunOS implements a different protocol that does support two-phase locking.

We also suggest more flexibility and extensibility in the following areas:

- *Fault Tolerance/Recovery*

The facility should provide support for recovery from file server failures. It should allow a processes to transfer locks.

- *Scheduling*

The facility should allow customization of the scheduler to support priority-based or semantic scheduling algorithms. Parameters should be added to `fcntl()` for timeout and sets of locks.

- *Nested Transactions*

The facility should support the definition of protocols for parent and child processes that differ from those that cover unrelated processes to allow a simple implementation of nested transactions.

9.2 Further study

A performance comparison with a commercial DBMS lock manager would be useful and enlightening. Unfortunately, it requires access to an internal, nonpublic interface in the DBMS, and as such, it requires the cooperation of the vendor. To date, we have not found a vendor interested in the comparison.

9.3 Conclusion

In conclusion, we are still very interested in the prospect of OS support for DBMSs. There is a

Properties	Rationale
They primarily lock files on the same machine.	Local locking is less costly than remote; deadlock is not detected across file systems.
They are mostly readers; writers are rare.	The scheduler is <i>Reader's priority</i> (caveat: writers to shared data may wait a long time).
The segments locked by different transactions do not overlap.	Starvation is possible when a transaction requests a lock on a file segment, part of which is locked in a conflicting mode by another transaction.
They lock file segments that are contiguous or randomly ordered.	Contiguous locking is inexpensive, and so is randomly ordered locking in some OSs.
Each transaction locks fewer files than its maximum number of open file descriptors.	The OS releases a transaction's locks on a file upon closing.
All transactions have equal priority.	The scheduler does not support priority-based schemes.

Table 2: Characteristics of transactions that are well-suited to OS locking

trend in OS research toward microkernel architectures that could allow the kind of customization and flexibility that DBMSs need. Moreover, there is interest in the OS research community to provide better support for applications. Finally, many of the problems we cite have solutions that are backward-compatible with the current interface (e.g., the *Read'* lock).

Acknowledgments

We thank Dan Lieuwen and Mike Zwilling for their help with the timing studies on EXODUS, and we especially thank the shepherd for her detailed comments and suggestions.

References

- [1] A. Protin, UNIX System Laboratories. Personal communication, April 1994.
 - [2] AT&T. *UNIX System V Release 4 Programmers Reference Manual*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1990.
 - [3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, 1987.
 - [4] M. Carey, D. DeWitt, J. Richardson, and E. Shekita. Object and File Management in the EXODUS Extensible Database System. In *Proceedings of Twelfth International Conference on Very Large Data Bases*, 1986.
 - [5] R. L. Greer. DataShare and the Fourth-Generation Language Cymbol. Internal Technical Report 59113-920901-12TMS, AT&T Bell Laboratories, September 1992.
 - [6] The Institute of Electrical and Electronics Engineers, Inc., New York, NY. *Information technology — Portable Operating System Interface (POSIX). Part 1: System Application Program Interface (API) [C Language]*, 1990.
 - [7] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. The MIT Press, Cambridge, MA, 1985.
 - [8] K. Ramamritham. Real-Time Databases. *Journal of Distributed and Parallel Databases*, 1(2), 1993.
 - [9] M. Seltzer. Personal communication, April 1994.
 - [10] M. Stonebraker, D. DuBourdieu, and W. Edwards. Problems in Supporting Data Base Transactions in an Operating System Transaction Manager. *Operating Systems Review*, 19(1), January 1985.
 - [11] Sun Microsystems, Inc., Mountain View, CA. *SunOS Reference Manual*, 1992.
- Andrea H. Skarra** (ahs@research.att.com) is a Member of Technical Staff in the Software and Systems Research Center, AT&T Bell Laboratories, Room 2B-101, Murray Hill NJ 07974. Her research interests lie primarily in the areas of semantics-based concurrency control and database integration and evolution.
- HP-UX is a trademark of Hewlett-Packard, Inc. RISC/os is a trademark of MIPS Computer Systems, Inc.
- IRIX is a trademark of Silicon Graphics, Inc.
- SunOS, Solaris, and NFS are trademarks of Sun Microsystems, Inc.
- UNIX is a registered trademark of UNIX System Laboratories.

The Slab Allocator: An Object-Caching Kernel Memory Allocator

Jeff Bonwick
Sun Microsystems

Abstract

This paper presents a comprehensive design overview of the SunOS 5.4 kernel memory allocator. This allocator is based on a set of object-caching primitives that reduce the cost of allocating complex objects by retaining their state between uses. These same primitives prove equally effective for managing stateless memory (e.g. data pages and temporary buffers) because they are space-efficient and fast. The allocator's object caches respond dynamically to global memory pressure, and employ an object-coloring scheme that improves the system's overall cache utilization and bus balance. The allocator also has several statistical and debugging features that can detect a wide range of problems throughout the system.

1. Introduction

The allocation and freeing of objects are among the most common operations in the kernel. A fast kernel memory allocator is therefore essential. However, in many cases the cost of initializing and destroying the object exceeds the cost of allocating and freeing memory for it. Thus, while improvements in the allocator are beneficial, even greater gains can be achieved by caching frequently used objects so that their basic structure is preserved between uses.

The paper begins with a discussion of object caching, since the interface that this requires will shape the rest of the allocator. The next section then describes the implementation in detail. Section 4 describes the effect of buffer address distribution on the system's overall cache utilization and bus balance, and shows how a simple coloring scheme can improve both. Section 5 compares the allocator's performance to several other well-known kernel memory allocators and finds that it is

generally superior in both space *and* time. Finally, Section 6 describes the allocator's debugging features, which can detect a wide variety of problems throughout the system.

2. Object Caching

Object caching is a technique for dealing with objects that are frequently allocated and freed. The idea is to preserve the invariant portion of an object's initial state — its *constructed* state — between uses, so it does not have to be destroyed and recreated every time the object is used. For example, an object containing a mutex only needs to have `mutex_init()` applied once — the first time the object is allocated. The object can then be freed and reallocated many times without incurring the expense of `mutex_destroy()` and `mutex_init()` each time. An object's embedded locks, condition variables, reference counts, lists of other objects, and read-only data all generally qualify as constructed state.

Caching is important because the cost of constructing an object can be significantly higher than the cost of allocating memory for it. For example, on a SPARCstation-2 running a SunOS 5.4 development kernel, the allocator presented here reduced the cost of allocating and freeing a stream head from 33 microseconds to 5.7 microseconds. As the table below illustrates, most of the savings was due to object caching:

Stream Head Allocation + Free Costs (μsec)			
allocator	construction + destruction	memory allocation	other init.
old	23.6	9.4	1.9
new	0.0	3.8	1.9

Caching is particularly beneficial in a multithreaded environment, where many of the most

frequently allocated objects contain one or more embedded locks, condition variables, and other constructible state.

The design of an object cache is straightforward:

To allocate an object:

```
if (there's an object in the cache)
    take it (no construction required);
else {
    allocate memory;
    construct the object;
}
```

To free an object:

```
return it to the cache (no destruction required);
```

To reclaim memory from the cache:

```
take some objects from the cache;
destroy the objects;
free the underlying memory;
```

An object's constructed state must be initialized only once — when the object is first brought into the cache. Once the cache is populated, allocating and freeing objects are fast, trivial operations.

2.1. An Example

Consider the following data structure:

```
struct foo {
    kmutex_t    foo_lock;
    kcondvar_t  foo_cv;
    struct bar *foo_barlist;
    int         foo_refcnt;
};
```

Assume that a foo structure cannot be freed until there are no outstanding references to it (`foo_refcnt == 0`) and all of its pending bar events (whatever they are) have completed (`foo_barlist == NULL`). The life cycle of a dynamically allocated foo would be something like this:

```
foo = kmalloc(sizeof (struct foo),
              KM_SLEEP);
mutex_init(&foo->foo_lock, ...);
cv_init(&foo->foo_cv, ...);
foo->foo_refcnt = 0;
foo->foo_barlist = NULL;

use foo;

ASSERT(foo->foo_barlist == NULL);
ASSERT(foo->foo_refcnt == 0);
cv_destroy(&foo->foo_cv);
mutex_destroy(&foo->foo_lock);
kmem_free(foo);
```

Notice that between each use of a foo object we perform a sequence of operations that constitutes nothing more than a very expensive no-op. All of this overhead (i.e., everything other than “use foo” above) can be eliminated by object caching.

2.2. The Case for Object Caching in the Central Allocator

Of course, object caching can be implemented without any help from the central allocator — any subsystem can have a private implementation of the algorithm described above. However, there are several disadvantages to this approach:

- (1) There is a natural tension between an object cache, which wants to keep memory, and the rest of the system, which wants that memory back. Privately-managed caches cannot handle this tension sensibly. They have limited insight into the system's overall memory needs and *no* insight into each other's needs. Similarly, the rest of the system has no knowledge of the existence of these caches and hence has no way to “pull” memory from them.
- (2) Since private caches bypass the central allocator, they also bypass any accounting mechanisms and debugging features that allocator may possess. This makes the operating system more difficult to monitor and debug.
- (3) Having many instances of the same solution to a common problem increases kernel code size and maintenance costs.

Object caching requires a greater degree of cooperation between the allocator and its clients than the standard `kmalloc(9F)`/`kmem_free(9F)` interface allows. The next section develops an interface to support constructed object caching in the central allocator.

2.3. Object Cache Interface

The interface presented here follows from two observations:

- (A) Descriptions of objects (name, size, alignment, constructor, and destructor) belong in the clients — not in the central allocator. The allocator should not just “know” that `sizeof (struct inode)` is a useful pool size, for example. Such assumptions are brittle [Grunwald93A] and cannot anticipate the needs of third-party device drivers, streams modules and file systems.
- (B) Memory management policies belong in the central allocator — not in its clients. The clients just want to allocate and free objects quickly. They shouldn’t have to worry about how to manage the underlying memory efficiently.

It follows from (A) that object cache creation must be client-driven and must include a full specification of the objects:

```
(1) struct kmem_cache *kmem_cache_create(
    char *name,
    size_t size,
    int align,
    void (*constructor)(void *, size_t),
    void (*destructor)(void *, size_t));
```

Creates a cache of objects, each of size `size`, aligned on an `align` boundary. The alignment will always be rounded up to the minimum allowable value, so `align` can be zero whenever no special alignment is required. `name` identifies the cache for statistics and debugging. `constructor` is a function that constructs (that is, performs the one-time initialization of) objects in the cache; `destructor` undoes this, if applicable. The constructor and destructor take a `size` argument so that they can support families of similar caches, e.g. streams messages. `kmem_cache_create` returns an opaque descriptor for accessing the cache.

Next, it follows from (B) that clients should need just two simple functions to allocate and free objects:

```
(2) void *kmem_cache_alloc(
    struct kmem_cache *cp,
    int flags);
```

Gets an object from the cache. The object will be in its constructed state. `flags` is either `KM_SLEEP` or `KM_NOSLEEP`, indicating

whether it’s acceptable to wait for memory if none is currently available.

```
(3) void kmem_cache_free(
    struct kmem_cache *cp,
    void *buf);
```

Returns an object to the cache. The object must still be in its constructed state.

Finally, if a cache is no longer needed the client can destroy it:

```
(4) void kmem_cache_destroy(
    struct kmem_cache *cp);
```

Destroys the cache and reclaims all associated resources. All allocated objects must have been returned to the cache.

This interface allows us to build a flexible allocator that is ideally suited to the needs of its clients. In this sense it is a “custom” allocator. However, it does not have to be built with compile-time knowledge of its clients as most custom allocators do [Bozman84A, Grunwald93A, Margolin71], nor does it have to keep guessing as in the adaptive-fit methods [Bozman84B, Leverett82, Oldehoeft85]. Rather, the object-cache interface allows clients to specify the allocation services they need on the fly.

2.4. An Example

This example demonstrates the use of object caching for the “foo” objects introduced in Section 2.1. The constructor and destructor routines are:

```
void
foo_constructor(void *buf, int size)
{
    struct foo *foo = buf;

    mutex_init(&foo->foo_lock, ...);
    cv_init(&foo->foo_cv, ...);
    foo->foo_refcnt = 0;
    foo->foo_barlist = NULL;
}

void
foo_destructor(void *buf, int size)
{
    struct foo *foo = buf;

    ASSERT(foo->foo_barlist == NULL);
    ASSERT(foo->foo_refcnt == 0);
    cv_destroy(&foo->foo_cv);
    mutex_destroy(&foo->foo_lock);
}
```

To create the foo cache:

```
foo_cache = kmem_cache_create("foo_cache",
    sizeof (struct foo), 0,
    foo_constructor, foo_destructor);
```

To allocate, use, and free a foo object:

```
foo = kmem_cache_alloc(foo_cache, KM_SLEEP);
use foo;
kmem_cache_free(foo_cache, foo);
```

This makes foo allocation fast, because the allocator will usually do nothing more than fetch an already-constructed foo from the cache. `foo_constructor` and `foo_destructor` will be invoked only to populate and drain the cache, respectively.

The example above illustrates a beneficial side-effect of object caching: it reduces the instruction-cache footprint of the code that *uses* cached objects by moving the rarely-executed construction and destruction code out of the hot path.

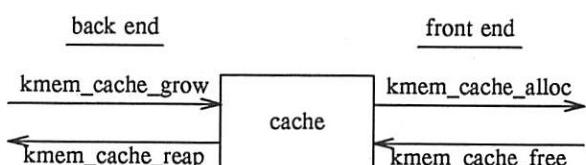
3. Slab Allocator Implementation

This section describes the implementation of the SunOS 5.4 kernel memory allocator, or “slab allocator,” in detail. (The name derives from one of the allocator’s main data structures, the *slab*. The name stuck within Sun because it was more distinctive than “object” or “cache.” Slabs will be discussed in Section 3.2.)

The terms *object*, *buffer*, and *chunk* will be used more or less interchangeably, depending on how we’re viewing that piece of memory at the moment.

3.1. Caches

Each cache has a *front end* and *back end* which are designed to be as decoupled as possible:



The front end is the public interface to the allocator. It moves objects to and from the cache, calling into the back end when it needs more objects.

The back end manages the flow of real memory through the cache. The influx routine

(`kmem_cache_grow()`) gets memory from the VM system, makes objects out of it, and feeds those objects into the cache. The outflux routine (`kmem_cache_reap()`) is invoked by the VM system when it wants some of that memory back — e.g., at the onset of paging. Note that all back-end activity is triggered solely by memory pressure. Memory flows in when the cache needs more objects and flows back out when the rest of the system needs more pages; there are no arbitrary limits or watermarks. Hysteresis control is provided by a working-set algorithm, described in Section 3.4.

The slab allocator is not a monolithic entity, but rather is a loose confederation of independent caches. The caches have no shared state, so the allocator can employ per-cache locking instead of protecting the entire arena (kernel heap) with one global lock. Per-cache locking improves scalability by allowing any number of distinct caches to be accessed simultaneously.

Each cache maintains its own statistics — total allocations, number of allocated and free buffers, etc. These per-cache statistics provide insight into overall system behavior. They indicate which parts of the system consume the most memory and help to identify memory leaks. They also indicate the activity level in various subsystems, to the extent that allocator traffic is an accurate approximation. (Streams message allocation is a direct measure of streams activity, for example.)

The slab allocator is operationally similar to the “CustoMalloc” [Grunwald93A], “QuickFit” [Weinstock88], and “Zone” [VanSciver88] allocators, all of which maintain distinct freelists of the most commonly requested buffer sizes. The Grunwald and Weinstock papers each demonstrate that a customized segregated-storage allocator — one that has *a priori* knowledge of the most common allocation sizes — is usually optimal in both space *and* time. The slab allocator is in this category, but has the advantage that its customizations are client-driven at run time rather than being hard-coded at compile time. (This is also true of the Zone allocator.)

The standard non-caching allocation routines, `kmem_alloc(9F)` and `kmem_free(9F)`, use object caches internally. At startup, the system creates a set of about 30 caches ranging in size from 8 bytes to 9K in roughly 10-20% increments. `kmem_alloc()` simply performs a `kmem_cache_alloc()` from the nearest-size cache. Allocations larger than 9K, which are rare, are handled directly by the back-end page supplier.

3.2. Slabs

The *slab* is the primary unit of currency in the slab allocator. When the allocator needs to grow a cache, for example, it acquires an entire slab of objects at once. Similarly, the allocator reclaims unused memory (shrinks a cache) by relinquishing a complete slab.

A slab consists of one or more pages of virtually contiguous memory carved up into equal-size chunks, with a reference count indicating how many of those chunks have been allocated. The benefits of using this simple data structure to manage the arena are somewhat striking:

(1) **Reclaiming unused memory is trivial.** When the slab reference count goes to zero the associated pages can be returned to the VM system. Thus a simple reference count replaces the complex trees, bitmaps, and coalescing algorithms found in most other allocators [Knuth68, Korn85, Standish80].

(2) **Allocating and freeing memory are fast, constant-time operations.** All we have to do is move an object to or from a freelist and update a reference count.

(3) **Severe external fragmentation (unused buffers on the freelist) is unlikely.** Over time, many allocators develop an accumulation of small, unusable buffers. This occurs as the allocator splits existing free buffers to satisfy smaller requests. For example, the right sequence of 32-byte and 40-byte allocations may result in a large accumulation of free 8-byte buffers — even though no 8-byte buffers are ever requested [Standish80]. A segregated-storage allocator cannot suffer this fate, since the only way to populate its 8-byte freelist is to actually allocate and free 8-byte buffers. Any sequence of 32-byte and 40-byte allocations — no matter how complex — can only result in population of the 32-byte and 40-byte freelists. Since prior allocation is a good predictor of future allocation [Weinstock88] these buffers are likely to be used again.

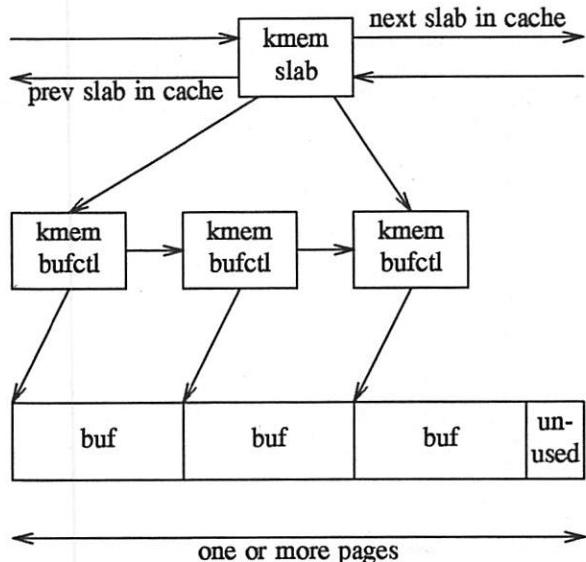
The other reason that slabs reduce external fragmentation is that all objects in a slab are of the same type, so they have the same lifetime distribution.* The resulting segregation of short-lived and long-lived objects at slab granularity reduces the likelihood of an entire page being held hostage due to a single long-lived allocation [Barrett93, Hanson90].

(4) **Internal fragmentation (per-buffer wasted space) is minimal.** Each buffer is exactly the right size (namely, the cache's object size), so the only wasted space is the unused portion at the end of the slab. For example, assuming 4096-byte pages, the slabs in a 400-byte object cache would each contain 10 buffers, with 96 bytes left over. We can view this as equivalent 9.6 bytes of wasted space per 400-byte buffer, or 2.4% internal fragmentation.

In general, if a slab contains n buffers, then the internal fragmentation is at most $1/n$; thus the allocator can actually control the amount of internal fragmentation by controlling the slab size. However, larger slabs are more likely to cause external fragmentation, since the probability of being able to reclaim a slab decreases as the number of buffers per slab increases. The SunOS 5.4 implementation limits internal fragmentation to 12.5% (1/8), since this was found to be the empirical sweet-spot in the trade-off between internal and external fragmentation.

3.2.1. Slab Layout — Logical

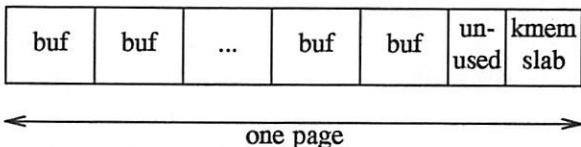
The contents of each slab are managed by a `kmem_slab` data structure that maintains the slab's linkage in the cache, its reference count, and its list of free buffers. In turn, each buffer in the slab is managed by a `kmem_bufctl` structure that holds the freelist linkage, buffer address, and a back-pointer to the controlling slab. Pictorially, a slab looks like this (bufctl-to-slab back-pointers not shown):



* The generic caches that back `kmem_alloc()` are a notable exception, but they constitute a relatively small fraction of the arena in SunOS 5.4 — all of the major consumers of memory now use `kmem_cache_alloc()`.

3.2.2. Slab Layout for Small Objects

For objects smaller than 1/8 of a page, a slab is built by allocating a page, placing the slab data at the end, and dividing the rest into equal-size buffers:



Each buffer serves as its own bufctl while on the freelist. Only the linkage is actually needed, since everything else is computable. These are essential optimizations for small buffers — otherwise we would end up allocating almost as much memory for bufctls as for the buffers themselves.

The freelist linkage resides at the *end* of the buffer, rather than the beginning, to facilitate debugging. This is driven by the empirical observation that the beginning of a data structure is typically more active than the end. If a buffer is modified after being freed, the problem is easier to diagnose if the heap *structure* (freelist linkage) is still intact.

The allocator reserves an additional word for constructed objects so that the linkage doesn't overwrite any constructed state.

3.2.3. Slab Layout for Large Objects

The above scheme is efficient for small objects, but not for large ones. It could fit only *one* 2K buffer on a 4K page because of the embedded slab data. Moreover, with large (multi-page) slabs we lose the ability to determine the slab data address from the buffer address. Therefore, for large objects the physical layout is identical to the logical layout. The required slab and bufctl data structures come from their own (small-object!) caches. A per-cache self-scaling hash table provides buffer-to-bufctl conversion.

3.3. Freelist Management

Each cache maintains a circular, doubly-linked list of all its slabs. The slab list is partially sorted, in that the empty slabs (all buffers allocated) come first, followed by the partial slabs (some buffers allocated, some free), and finally the complete slabs (all buffers free, refcnt == 0). The cache's freelist pointer points to its first non-empty slab. Each slab, in turn, has its own freelist of available buffers. This two-level freelist structure simplifies memory

reclaiming. When the allocator reclaims a slab it doesn't have to unlink each buffer from the cache's freelist — it just unlinks the slab.

3.4. Reclaiming Memory

When `kmem_cache_free()` sees that the slab reference count is zero, it does not immediately reclaim the memory. Instead, it just moves the slab to the tail of the freelist where all the complete slabs reside. This ensures that no complete slab will be broken up unless all partial slabs have been depleted.

When the system runs low on memory it asks the allocator to liberate as much memory as it can. The allocator obliges, but retains a 15-second working set of recently-used slabs to prevent thrashing. Measurements indicate that system performance is fairly insensitive to the slab working-set interval. Presumably this is because the two extremes — zero working set (reclaim all complete slabs on demand) and infinite working-set (never reclaim anything) — are both reasonable, albeit suboptimal, policies.

4. Hardware Cache Effects

Modern hardware relies on good cache utilization, so it is important to design software with cache effects in mind. For a memory allocator there are two broad classes of cache effects to consider: the distribution of buffer addresses and the cache footprint of the allocator itself. The latter topic has received some attention [Chen93, Grunwald93B], but the effect of buffer address distribution on cache utilization and bus balance has gone largely unrecognized.

4.1. Impact of Buffer Address Distribution on Cache Utilization

The address distribution of mid-size buffers can affect the system's overall cache utilization. In particular, power-of-two allocators — where all buffers are 2^n bytes and are 2^n -byte aligned — are pessimistic.* Suppose, for example, that every inode (~ 300 bytes) is assigned a 512-byte buffer, 512-byte aligned, and that only the first dozen fields of an inode (48 bytes) are frequently referenced. Then the majority of inode-related memory traffic will be

* Such allocators are common because they are easy to implement. For example, 4.4BSD and SVr4 both employ power-of-two methods [McKusick88, Lee89].

at addresses between 0 and 47 modulo 512. Thus the cache lines near 512-byte boundaries will be heavily loaded while the rest lie fallow. In effect only 9% (48/512) of the cache will be usable by inodes. Fully-associative caches would not suffer this problem, but current hardware trends are toward simpler rather than more complex caches.

Of course, there's nothing special about inodes. The kernel contains many other mid-size data structures (e.g. 100-500 bytes) with the same essential qualities: there are many of them, they contain only a few heavily used fields, and those fields are grouped together at or near the beginning of the structure. This artifact of the way data structures evolve has not previously been recognized as an important factor in allocator design.

4.2. Impact of Buffer Address Distribution on Bus Balance

On a machine that interleaves memory across multiple main buses, the effects described above also have a significant impact on bus utilization. The SPARCcenter 2000, for example, employs 256-byte interleaving across two main buses [Cekleov92]. Continuing the example above, we see that any power-of-two allocator maps the first half of every inode (the hot part) to bus 0 and the second half to bus 1. Thus almost all inode-related cache misses are serviced by bus 0. The situation is exacerbated by an inflated miss rate, since all of the inodes are fighting over a small fraction of the cache.

These effects can be dramatic. On a SPARCcenter 2000 running LADDIS under a SunOS 5.4 development kernel, replacing the old allocator (a power-of-two buddy-system [Lee89]) with the slab allocator reduced bus imbalance from 43% to just 17%. In addition, the primary cache miss rate dropped by 13%.

4.3. Slab Coloring

The slab allocator incorporates a simple coloring scheme that distributes buffers evenly throughout the cache, resulting in excellent cache utilization and bus balance. The concept is simple: each time a new slab is created, the buffer addresses start at a slightly different offset (color) from the slab base (which is always page-aligned). For example, for a cache of 200-byte objects with 8-byte alignment, the first slab's buffers would be at addresses 0, 200, 400, ... relative to the slab base. The next slab's buffers would be at offsets 8, 208, 408, ... and so

on. The maximum slab color is determined by the amount of unused space in the slab. In this example, assuming 4K pages, we can fit 20 200-byte buffers in a 4096-byte slab. The buffers consume 4000 bytes, the `kmem_slab` data consumes 32 bytes, and the remaining 64 bytes are available for coloring. Thus the maximum slab color is 64, and the slab color sequence is 0, 8, 16, 24, 32, 40, 48, 56, 64, 0, 8, ...

One particularly nice property of this coloring scheme is that mid-size power-of-two buffers receive the maximum amount of coloring, since they are the worst-fitting. For example, while 128 bytes goes perfectly into 4096, it goes near-pessimally into 4096 - 32, which is what's actually available (because of the embedded slab data).

4.4. Arena Management

An allocator's arena management strategy determines its dynamic cache footprint. These strategies fall into three broad categories: sequential-fit methods, buddy methods, and segregated-storage methods [Standish80].

A sequential-fit allocator must typically search several nodes to find a good-fitting buffer. Such methods are, by nature, condemned to a large cache footprint: they have to examine a significant number of nodes that are generally nowhere near each other. This causes not only cache misses, but TLB misses as well. The coalescing stages of buddy-system allocators [Knuth68, Lee89] have similar properties.

A segregated-storage allocator, such as the slab allocator, maintains separate freelist for different buffer sizes. These allocators generally have good cache locality because allocating a buffer is so simple. All the allocator has to do is determine the right freelist (by computation, by table lookup, or by having it supplied as an argument) and take a buffer from it. Freeing a buffer is similarly straightforward. There are only a handful of pointers to load, so the cache footprint is small.

The slab allocator has the additional advantage that for small to mid-size buffers, most of the relevant information — the slab data, bufctls, and buffers themselves — resides on a single page. Thus a single TLB entry covers most of the action.

5. Performance

This section compares the performance of the slab allocator to three other well-known kernel memory allocators:

SunOS 4.1.3, based on [Stephenson83], a sequential-fit method;

4.4BSD, based on [McKusick88], a power-of-two segregated-storage method;

SVr4, based on [Lee89], a power-of-two buddy-system method. This allocator was employed in all previous SunOS 5.x releases.

To get a fair comparison, each of these allocators was ported into the same SunOS 5.4 base system. This ensures that we are comparing just allocators, not entire operating systems.

5.1. Speed Comparison

On a SPARCstation-2 the time required to allocate and free a buffer under the various allocators is as follows:

Memory Allocation + Free Costs		
allocator	time (μsec)	interface
slab	3.8	kmem_cache_alloc
4.4BSD	4.1	kmem_alloc
slab	4.7	kmem_alloc
SVr4	9.4	kmem_alloc
SunOS 4.1.3	25.0	kmem_alloc

Note: The 4.4BSD allocator offers both functional and preprocessor macro interfaces. These measurements are for the functional version. Non-binary interfaces in general were not considered, since these cannot be exported to drivers without exposing the implementation. The 4.4BSD allocator was compiled *without* KMEMSTATS defined (it's on by default) to get the fastest possible code.

A mutex_enter() /mutex_exit() pair costs 1.0 μsec, so the locking required to allocate and free a buffer imposes a lower bound of 2.0 μsec. The slab and 4.4BSD allocators are both very close to this limit because they do very little work in the common cases. The 4.4BSD implementation of kmem_alloc() is slightly faster, since it has less accounting to do (it never reclaims memory). The slab allocator's kmem_cache_alloc() interface is even faster, however, because it doesn't have to determine which freelist (cache) to use — the cache descriptor is passed as an argument to kmem_cache_alloc(). In any event, the differences in speed between the slab and 4.4BSD

allocators are small. This is to be expected, since all segregated-storage methods are operationally similar. Any good segregated-storage implementation should achieve excellent performance.

The SVr4 allocator is slower than most buddy systems but still provides reasonable, predictable speed. The SunOS 4.1.3 allocator, like most sequential-fit methods, is comparatively slow and quite variable.

The benefits of object caching are not visible in the numbers above, since they only measure the cost of the allocator itself. The table below shows the effect of object caching on some of the most frequent allocations in the SunOS 5.4 kernel (SPARCstation-2 timings, in microseconds):

Effect of Object Caching			
allocation type	without caching	with caching	improvement
allocb	8.3	6.0	1.4x
dupb	13.4	8.7	1.5x
shalloc	29.3	5.7	5.1x
allocq	40.0	10.9	3.7x
anonmap_alloc	16.3	10.1	1.6x
makepipe	126.0	98.0	1.3x

All of the numbers presented in this section measure the performance of the allocator in isolation. The allocator's effect on overall system performance will be discussed in Section 5.3.

5.2. Memory Utilization Comparison

An allocator generally consumes more memory than its clients actually request due to imperfect fits (internal fragmentation), unused buffers on the freelist (external fragmentation), and the overhead of the allocator's internal data structures. The ratio of memory requested to memory consumed is the allocator's *memory utilization*. The complementary ratio is the *memory wastage* or *total fragmentation*. Good memory utilization is essential, since the kernel heap consumes physical memory.

An allocator's space efficiency is harder to characterize than its speed because it is workload-dependent. The best we can do is to measure the various allocators' memory utilization under a fixed set of workloads. To this end, each allocator was subjected to the following workload sequence:

- (1) System boot. This measures the system's memory utilization at the console login prompt after rebooting.

- (2) A brief spike in load, generated by the following trivial program:

```
fork(); fork(); fork(); fork();
fork(); fork(); fork(); fork();
fd = socket(AF_UNIX, SOCK_STREAM, 0);
sleep(60);
close(fd);
```

This creates 256 processes, each of which creates a socket. This causes a temporary surge in demand for a variety of kernel data structures.

- (3) Find. This is another trivial spike-generator:

```
find /usr -mount -exec file {} \;
```

- (4) Kenbus. This is a standard timesharing benchmark. Kenbus generates a large amount of concurrent activity, creating large demand for both user and kernel memory.

Memory utilization was measured after each step. The table below summarizes the results for a 16MB SPARCstation-1. The slab allocator significantly outperformed the others, ending up with half the fragmentation of the nearest competitor (results are cumulative, so the “kenbus” column indicates the fragmentation after all four steps were completed):

	Total Fragmentation (waste)					
allocator	boot	spike	find	kenbus	s/m	
slab	11%	13%	14%	14%	233	
SunOS 4.1.3	7%	19%	19%	27%	210	
4.4BSD	20%	43%	43%	45%	205	
SVr4	23%	45%	45%	46%	199	

The last column shows the kenbus results, which measure peak throughput in units of scripts executed per minute (s/m). Kenbus performance is primarily memory-limited on this 16MB system, which is why the SunOS 4.1.3 allocator achieved better results than the 4.4BSD allocator despite being significantly slower. The slab allocator delivered the best performance by an 11% margin because it is both fast *and* space-efficient.

To get a handle on real-life performance the author used each of these allocators for a week on his personal desktop machine, a 32MB SPARCstation-2. This machine is primarily used for reading e-mail, running simple commands and scripts, and connecting to test machines and compute servers. The results of this obviously non-controlled experiment were:

Effect of One Week of Light Desktop Use		
allocator	kernel heap	fragmentation
slab	6.0 MB	9%
SunOS 4.1.3	6.7 MB	17%
SVr4	8.5 MB	35%
4.4BSD	9.0 MB	38%

These numbers are consistent with the results from the synthetic workload described above. In both cases, the slab allocator generates about half the fragmentation of SunOS 4.1.3, which in turn generates about half the fragmentation of SVr4 and 4.4BSD.

5.3. Overall System Performance

The kernel memory allocator affects overall system performance in a variety of ways. In previous sections we considered the effects of several individual factors: object caching, hardware cache and bus effects, speed, and memory utilization. We now turn to the most important metric: the bottom-line performance of interesting workloads. In SunOS 5.4 the SVr4-based allocator was replaced by the slab allocator described here. The table below shows the net performance improvement in several key areas.

System Performance Improvement with Slab Allocator		
workload	gain	what it measures
DeskBench	12%	window system
kenbus	17%	timesharing
TPC-B	4%	database
LADDIS	3%	NFS service
parallel make	5%	parallel compilation
terminal server	5%	many-user typing

Notes:

- (1) DeskBench and kenbus are both memory-bound in 16MB, so most of the improvement here is due to the slab allocator’s space efficiency.
- (2) The TPC-B workload causes very little kernel memory allocation, so the allocator’s speed is not a significant factor here. The test was run on a large server with enough memory that it never paged (under either allocator), so space efficiency is not a factor either. The 4% performance improvement is due solely to better cache utilization (5% fewer primary cache misses, 2% fewer secondary cache misses).

- (3) Parallel make was run on a large server that never paged. This workload generates a lot of allocator traffic, so the improvement here is attributable to the slab allocator's speed, object caching, and the system's lower overall cache miss rate (5% fewer primary cache misses, 4% fewer secondary cache misses).
- (4) Terminal server was also run on a large server that never paged. This benchmark spent 25% of its time in the kernel with the old allocator, versus 20% with the new allocator. Thus, the 5% bottom-line improvement is due to a 20% reduction in kernel time.

6. Debugging Features

Programming errors that corrupt the kernel heap — such as modifying freed memory, freeing a buffer twice, freeing an uninitialized pointer, or writing beyond the end of a buffer — are often difficult to debug. Fortunately, a thoroughly instrumented kernel memory allocator can detect many of these problems.

This section describes the debugging features of the slab allocator. These features can be enabled in any SunOS 5.4 kernel (not just special debugging versions) by booting under kadb (the kernel debugger) and setting the appropriate flags.* When the allocator detects a problem, it provides detailed diagnostic information on the system console.

6.1. Auditing

In audit mode the allocator records its activity in a circular transaction log. It stores this information in an extended version of the bufctl structure that includes the thread pointer, hi-res timestamp, and stack trace of the transaction. When corruption is detected by any of the other methods, the previous owners of the affected buffer (the likely suspects) can be determined.

6.2. Freed-Address Verification

The buffer-to-bufctl hash table employed by large-object caches can be used as a debugging feature: if

* The availability of these debugging features adds no cost to most allocations. The per-cache flag word that indicates whether a hash table is present — i.e., whether the cache's objects are larger than 1/8 of a page — also contains the debugging flags. A single test checks all of these flags simultaneously, so the common case (small objects, no debugging) is unaffected.

the hash lookup in `kmem_cache_free()` fails, then the caller must be attempting to free a bogus address. The allocator can verify *all* freed addresses by changing the “large object” threshold to zero.

6.3. Detecting Use of Freed Memory

When an object is freed, the allocator applies its destructor and fills it with the pattern 0xdeadbeef. The next time that object is allocated, the allocator verifies that it still contains the deadbeef pattern. It then fills the object with 0xbaddcafe and applies its constructor. The deadbeef and baddcafe patterns are chosen to be readily human-recognizable in a debugging session. They represent freed memory and uninitialized data, respectively.

6.4. Redzone Checking

Redzone checking detects writes past the end of a buffer. The allocator checks for redzone violations by adding a guard word to the end of each buffer and verifying that it is unmodified when the buffer is freed.

6.5. Synchronous Unmapping

Normally, the slab working-set algorithm retains complete slabs for a while. In synchronous-unmapping mode the allocator destroys complete slabs immediately. `kmem_slab_destroy()` returns the underlying memory to the back-end page supplier, which unmaps the page(s). Any subsequent reference to any object in that slab will cause a kernel data fault.

6.6. Page-per-buffer Mode

In page-per-buffer mode each buffer is given an entire page (or pages) so that every buffer can be unmapped when it is freed. The slab allocator implements this by increasing the alignment for all caches to the system page size. (This feature requires an obscene amount of physical memory.)

6.7. Leak Detection

The timestamps provided by auditing make it easy to implement a crude kernel memory leak detector at user level. All the user-level program has to do is periodically scan the arena (via `/dev/kmem`), looking for the appearance of new, persistent allocations. For example, any buffer that was allocated an hour ago and is still allocated now is a possible leak.

6.8. An Example

This example illustrates the slab allocator's response to modification of a free node:

```
kernel memory allocator: buffer modified after being freed  
modification occurred at offset 0x18 (0xdeadbeef replaced by 0x34)  
buffer=ff8eea20 bufctl=ff8efef0 cache: snode_cache  
previous transactions on buffer ff8eea20:  
  
thread=ff8b93a0 time=T-0.000089 slab=ff8ca8c0 cache: snode_cache  
kmem_cache_alloc+f8  
specvp+48  
ufs_lookup+148  
lookuppn+3ac  
lookupname+28  
vn_open+a4  
copen+6c  
syscall+3e8  
  
thread=ff8b94c0 time=T-1.830247 slab=ff8ca8c0 cache: snode_cache  
kmem_cache_free+128  
spec_inactive+208  
closef+94  
syscall+3e8  
  
(transaction log continues at ff31f410)  
kadb[0]:
```

Other errors are handled similarly. These features have proven helpful in debugging a wide range of problems during SunOS 5.4 development.

7. Future Directions

7.1. Managing Other Types of Memory

The slab allocator gets its pages from segkmem via the routines `kmem_getpages()` and `kmem_freepages()`; it assumes nothing about the underlying segment driver, resource maps, translation setup, etc. Since the allocator respects this firewall, it would be trivial to plug in alternate back-end page suppliers. The “getpages” and “freepages” routines could be supplied as additional arguments to `kmem_cache_create()`. This would allow us to manage multiple types of memory (e.g. normal kernel memory, device memory, pageable kernel memory, NVRAM, etc.) with a single allocator.

7.2. Per-Processor Memory Allocation

The per-processor allocation techniques of McKenney and Slingwine [McKenney93] would fit nicely on top of the slab allocator. They define a four-layer allocation hierarchy of decreasing speed and locality: per-CPU, global, coalesce-to-page, and coalesce-to-VM-block. The latter three correspond closely to the slab allocator's front-end, back-end, and page-supplier layers, respectively. Even in the

absence of lock contention, small per-processor freelists could improve performance by eliminating locking costs and reducing invalidation traffic.

7.3. User-level Applications

The slab allocator could also be used as a user-level memory allocator. The back-end page supplier could be `mmap(2)` or `sbrk(2)`.

8. Conclusions

The slab allocator is a simple, fast, and space-efficient kernel memory allocator. The object-cache interface upon which it is based reduces the cost of allocating and freeing complex objects and enables the allocator to segregate objects by size and lifetime distribution. Slabs take advantage of object size and lifetime segregation to reduce internal and external fragmentation, respectively. Slabs also simplify reclaiming by using a simple reference count instead of coalescing. The slab allocator establishes a push/pull relationship between its clients and the VM system, eliminating the need for arbitrary limits or watermarks to govern reclaiming. The allocator's coloring scheme distributes buffers evenly throughout the cache, improving the system's overall cache utilization and bus balance. In several important areas, the slab allocator provides measurably better system performance.

Acknowledgements

Neal Nuckolls first suggested that the allocator should retain an object's state between uses, as our old streams allocator did (it now uses the slab allocator directly). Steve Kleiman suggested using VM pressure to regulate reclaiming. Gordon Irlam pointed out the negative effects of power-of-two alignment on cache utilization; Adrian Cockcroft hypothesized that this might explain the bus imbalance we were seeing on some machines (it did).

I'd like to thank Cathy Bonwick, Roger Faulkner, Steve Kleiman, Tim Marsland, Rob Pike, Andy Roach, Bill Shannon, and Jim Voll for their thoughtful comments on draft versions of this paper. Thanks also to David Robinson, Chaitanya Tikku, and Jim Voll for providing some of the measurements, and to Ashok Singhal for providing the tools to measure cache and bus activity.

Most of all, I thank Cathy for putting up with me (and without me) during this project.

References

- [Barrett93] David A. Barrett and Benjamin G. Zorn, *Using Lifetime Predictors to Improve Memory Allocation Performance*. Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation, pp. 187-196 (1993).
- [Boehm88] H. Boehm and M. Weiser, *Garbage Collection in an Uncooperative Environment*. Software - Practice and Experience, v. 18, no. 9, pp 807-820 (1988).
- [Bozman84A] G. Bozman, W. Buco, T. Daly, and W. Tetzlaff, *Analysis of Free Storage Algorithms -- Revisited*. IBM Systems Journal, v. 23, no. 1, pp. 44-64 (1984).
- [Bozman84B] G. Bozman, *The Software Lookaside Buffer Reduces Search Overhead with Linked Lists*. Communications of the ACM, v. 27, no. 3, pp. 222-227 (1984).
- [Cekleov92] Michel Cekleov, Jean-Marc Frailong and Pradeep Sindhu, *Sun-4D Architecture*. Revision 1.4, 1992.
- [Chen93] J. Bradley Chen and Brian N. Bershad, *The Impact of Operating System Structure on Memory System Performance*. Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles, v. 27, no. 5, pp. 120-133 (1993).
- [Grunwald93A] Dirk Grunwald and Benjamin Zorn, *CustoMalloc: Efficient Synthesized Memory Allocators*. Software - Practice and Experience, v. 23, no. 8, pp. 851-869 (1993).
- [Grunwald93B] Dirk Grunwald, Benjamin Zorn and Robert Henderson, *Improving the Cache Locality of Memory Allocation*. Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation, pp. 177-186 (1993).
- [Hanson90] David R. Hanson, *Fast Allocation and Deallocation of Memory Based on Object Lifetimes*. Software - Practice and Experience, v. 20, no. 1, pp. 5-12 (1990).
- [Knuth68] Donald E. Knuth, *The Art of Computer Programming, Vol 1, Fundamental Algorithms*. Addison-Wesley, Reading, MA, 1968.
- [Korn85] David G. Korn and Kiem-Phong Vo, *In Search of a Better Malloc*. Proceedings of the Summer 1985 Usenix Conference, pp. 489-506.
- [Lee89] T. Paul Lee and R. E. Barkley, *A Watermark-based Lazy Buddy System for Kernel Memory Allocation*. Proceedings of the Summer 1989 Usenix Conference, pp. 1-13.
- [Leverett82] B. W. Leverett and P. G. Hibbard, *An Adaptive System for Dynamic Storage Allocation*. Software - Practice and Experience, v. 12, no. 3, pp. 543-555 (1982).
- [Margolin71] B. Margolin, R. Parmelee, and M. Schatzoff, *Analysis of Free Storage Algorithms*. IBM Systems Journal, v. 10, no. 4, pp. 283-304 (1971).
- [McKenney93] Paul E. McKenney and Jack Slingwine, *Efficient Kernel Memory Allocation on Shared-Memory Multiprocessors*. Proceedings of the Winter 1993 Usenix Conference, pp. 295-305.
- [McKusick88] Marshall Kirk McKusick and Michael J. Karels, *Design of a General Purpose Memory Allocator for the 4.3BSD UNIX Kernel*. Proceedings of the Summer 1988 Usenix Conference, pp. 295-303.
- [Oldehoeft85] Rodney R. Oldehoeft and Stephen J. Allan, *Adaptive Exact-Fit Storage Management*. Communications of the ACM, v. 28, pp. 506-511 (1985).
- [Standish80] Thomas Standish, *Data Structure Techniques*. Addison-Wesley, Reading, MA, 1980.
- [Stephenson83] C. J. Stephenson, *Fast Fits: New Methods for Dynamic Storage Allocation*. Proceedings of the Ninth ACM Symposium on Operating Systems Principles, v. 17, no. 5, pp. 30-32 (1983).
- [VanSciver88] James Van Sciver and Richard F. Rashid, *Zone Garbage Collection*. Proceedings of the Summer 1990 Usenix Mach Workshop, pp. 1-15.
- [Weinstock88] Charles B. Weinstock and William A. Wulf, *QuickFit: An Efficient Algorithm for Heap Storage Allocation*. ACM SIGPLAN Notices, v. 23, no. 10, pp. 141-144 (1988).
- [Zorn93] Benjamin Zorn, *The Measured Cost of Conservative Garbage Collection*. Software - Practice and Experience, v. 23, no. 7, pp. 733-756 (1993).

Author Information

Jeff Bonwick is a kernel hacker at Sun. He likes to rip out big, slow, old code and replace it with small, fast, new code. He still can't believe he gets paid for this. The author received a B.S. in Mathematics from the University of Delaware (1987) and an M.S. in Statistics from Stanford (1990). He can be flamed electronically at bonwick@eng.sun.com.

A Better Update Policy

Jeffrey C. Mogul

Digital Equipment Corporation Western Research Laboratory

Abstract

Some file systems can delay writing modified data to disk, in order to reduce disk traffic and overhead. Prudence dictates that such delays be bounded, in case the system crashes. We refer to an algorithm used to decide when to write delayed data back to disk as an *update policy*. Traditional UNIX® systems use a *periodic update* policy, writing back all delayed-write data once every 30 seconds. Periodic update is easy to implement but performs quite badly in some cases. This paper describes an *approximate* implementation of an *interval periodic update* policy, in which each individual delayed-write block is written when its age reaches a threshold. Interval periodic update adds little code to the kernel and can perform much better than periodic update. In particular, interval periodic update can avoid the huge variances in read response time caused by using periodic update with a large buffer cache.

1. Introduction

File systems usually cache data and meta-data in a main memory *buffer cache*, in order to improve performance. When a modification is made, the file system may write the new information to stable storage (e.g., disk) immediately, or it may delay the write. This leads to a tradeoff: delaying writes reduces the load on the disk and system overhead, but the data could be lost if the system crashes before the write occurs. In many cases, users can tolerate this vulnerability, and welcome the performance advantages of delayed writes.

UNIX® systems have traditionally supported delayed writes, from the earliest C-language version [13, 14] up through 4.3BSD [6] and its derivatives. In these systems, a modification of a

partially-filled data block results in a delayed write, while a modification that fills a block results in an immediate, although asynchronous, write. The ULTRIX™ operating system can be configured to be even more aggressive, delaying all writes of modified data.

Without some bound on the age of a delayed-write block, a system crash could cause loss of arbitrary data. Users would not tolerate this, so the file system does push delayed-write data out to disk, after a while. We use the term *update policy* to describe the algorithm that decides what to write out, and when.

UNIX systems have traditionally used a simple *periodic update* (or ‘‘PU’’) policy: once every 30 seconds, all dirty blocks in the file system’s buffer cache are placed on the output queue for the appropriate disk. Recent analytical and simulation results, presented by Carson and Setia [2], showed that the PU policy actually performs worse in many cases than the write-through (WT) policy (in which all writes are immediate). Their analysis showed that PU causes increased mean response times for read operations; the results presented in this paper show that PU can also increase the variance in read response time.

Because so many systems use the now suspect PU policy, it seemed like a good idea to validate the results of Carson and Setia on actual systems; their analyses and simulations, while careful, had to use certain simplifying assumptions. Carson and Setia suggested that several other update policies would provide better performance, so it also seemed useful to implement one of these and measure its performance.

In this paper, after discussing the theoretical background in some more detail, I describe an implemen-

tation of the *interval periodic update* (IPU) policy. I also present the results of some simple measurements of PU and IPU made using an actual implementation, rather than a model. The results appear to bear out the basic conclusions of Carson and Setia. What I found is that although use of delayed writes can improve mean response time, combining delayed writes with periodic update increases the variance in response time, but using interval periodic update both the mean and variance are improved.

2. Theoretical background

In this section, I discuss the previous simulation study, the possible alternative update policies, the choice of an update interval, and how this problem will scale with changes in technology.

2.1. Carson and Setia's results for Periodic Update

The study done by Carson and Setia showed that the periodic update (PU) policy, while easily implemented, can perform quite badly. By dumping all the dirty blocks into the disk queue at once, PU can cause lengthy queueing delays. Latency-sensitive synchronous operations, such as file reads or synchronous writes, are forced to wait behind latency-insensitive asynchronous operations in the queue. If the system were to use a write-through (WT) policy instead, disk write operations would normally be spread out more over time, and the queues would be shorter.

Carson and Setia show that the relative performance of WT and PU, measured in terms of mean read-response time, depends on several parameters:

Read load

The ratio of read operation arrival rate to the rate that the disk can support. A read load of 1.0 is one that the disk could just barely keep up with, if no writes were done.

Write load

The ratio of write operation arrival rate to the rate that the disk can support.

Cache hit ratio for writes

The fraction of write operations that are satisfied by modifying already-dirty blocks in the buffer cache.

They expressed their results in tables showing, for a given read load and write load, what write-hit ratio PU requires in order to match or exceed the performance of WT. (Since WT causes disk writes to occur almost immediately, it cannot benefit from write hits in the cache.)

They found that:

- When the disk is not overloaded, "the cache must eliminate 80-90 % of all write accesses before the PU policy pays off."
- Under heavy loads, the PU policy gets some benefit from write-hits in the cache, and thus reduces the overall disk load. In cases where the WT policy would saturate the disk, the PU policy gives a lower mean read-response time.

Cache hit rates vary, depending on cache size, application, and replacement policy, but we are unlikely to achieve average write-hit rates exceeding 80%. For example, Baker *et al.* [1] traced user-level file access patterns in a distributed system and found that 88% of the bytes written were written sequentially; if the applications in question used traditional buffering strategies, most of these sequential writes would not have hit already-dirty buffer-cache blocks, and so the write-hit rate must have been quite low.

Carson and Setia found an analytical model for the mean response time. Their simulations were used to validate this result, but they apparently did not investigate other statistics besides the mean. As section 4 will show, PU is especially bad for worst-case response time, and for overall variance in response time. It is not hard to construct a situation where PU can lead to worst-case read response times of many seconds.

2.2. Proposed alternative policies

PU performs poorly because it generates long queues at periodic intervals, and subsequent synchronous requests get stuck at the ends of these queues. Perhaps one could improve read-response times by changing the queueing mechanism.

UNIX systems typically maintain a single, unprioritized operation queue for each disk. Suppose that read operations were given priority over asynchronous operations already in the queue. Then, read operations would not "see" a queueing delay caused by the queued delayed writes. Carson and Setia analyzed this *periodic update with read priority* (PURP) policy, and showed that it mostly solves the read-response time problem. I found PURP unsatisfactory, however, for several reasons:

- Modification of the existing disk queue mechanism would require changes to numerous kernel modules, including all disk drivers and many of their clients (file systems, virtual memory systems, etc.)
- Modern disk controllers and drives can queue several operations in their internal buffers. One

would either have to accept the resulting queueing delays, or somehow modify the hardware to support the new queueing mechanism.

- Carson and Setia point out that fixed-priority schemes such as PURP introduce the potential for infinite delays of delayed writes, if the read load is enough to saturate the disk. Peacock [12] reported that adding PURP to System V Release 4 does seem to hurt benchmark performance, although it substantially increases single-file write throughput.

Although implementation of a prioritized queueing scheme should be helpful in general, it is neither a complete solution to the bursty-update problem, nor is it the simplest solution.

Carson and Setia also proposed the *interval periodic update* (IPU) policy, in which each dirty block is written out when its age reaches a threshold. If file modifications are nicely spread out in time, this means that the delayed writes back to the disk will also be spread out. As with PU and PURP, IPU uses the buffer cache to eliminate some disk writes that would be done by WT. Unlike PU and PURP, IPU normally avoids creating large bursts of writes, and so avoids the associated queueing delays. Carson and Setia show that IPU never gives worse mean read response time than WT or PU, although in some situations it may perform worse than PURP.

Anna Hac [3] describes algorithms meant for deciding when to move dirty blocks from the buffer cache to a disk queue. In essence these replace the time-driven update policies with dynamic algorithms, which choose when to schedule disk writes based on the system load and disk queue length. Such adaptive algorithms may perform better than any of the open-loop algorithms described in this paper, but they require more extensive changes to the operating system. I do not have anything useful to say about them, and they merit additional study.

2.3. Choice of update interval

UNIX systems have traditionally used a 30-second interval between writes generated by the PU policy. This means that, ignoring brief queueing delays, no information will be vulnerable to a crash for longer than 30 seconds. (Applications that depend on reliable data storage should arrange to write their data synchronously, using the *fsync()* system call. Many other applications, such as compilers, can afford to use delayed writes because their output can easily be reconstructed, or because if the system crashes during a run, the resulting partial output is useless anyway.)

Under the assumption that modifications occur more or less uniformly over time, the average age of a delayed-write block, when it is written to disk, is 15 seconds.

The IPU policy also has a characteristic time scale, the age at which a dirty buffer is scheduled for writing to the disk. If we set this to 30 seconds, then (again ignoring queueing delays), by definition, no information will be vulnerable to a crash for longer than 30 seconds. Also by definition, the average age of a block, when written to disk, is 30 seconds.

If we choose to set the update delay for IPU the same as the update interval for PU, then both policies expose modified data to exactly the same worst-case vulnerability. Doing so, however, means that the mean age of dirty blocks is roughly twice as it would be with the PU policy. This suggests that IPU should see a higher write-hit ratio, and might avoid a few more disk write operations.

Carson and Setia showed that, as the update interval was increased, the write-hit ratio at which PU began to pay off had to increase as well. We do expect this ratio to increase, but does it increase fast enough? Carson and Setia cite other work suggesting that it might [10] (see also a more recent study [1]). Unfortunately, I know of no actual test of this hypothesis. Still, one might suspect that the increased average lifetime of dirty blocks, when the IPU policy is used, might account for some performance advantage. (Note that the experiments reported in sections 4.1 and 4.2 carefully avoid repeated writes to the same block during an update interval, and so should encounter abnormally low cache hit ratios.)

2.4. Scaling properties

One might ask why, if PU performs so badly, has this not been a problem in practice¹. The answer is that buffer cache sizes and disk speeds are improving at different rates, which changes the ratio of disk queue length to disk service latency.

4.2BSD and related systems only delay partial-block writes. Since most files are written sequentially, most blocks are filled quickly, and pending delayed writes of partial blocks are turned into asynchronous immediate writes of filled blocks. If the buffer cache is not large enough to hold many entire files, then it makes little sense to delay writes

¹Some systems have indeed exhibited poor behavior resulting from disk queues containing many asynchronous write requests [12].

of full blocks, since these blocks are unlikely to be referenced again quickly.

Memory chips get larger: this is one of the most reliable laws of recent history. One can quibble over whether the doubling time is 18 months or two years, but main memory sizes do increase (at roughly constant cost) as the years pass, and no other technology trend is quite so steep [5].

This trend means that, if the fraction of main memory used as a buffer cache remains constant, the absolute size of buffer caches is increasing with time. (Many systems, including Mach, Sprite, and recent UNIX implementations, no longer allocate a fixed fraction of main memory for the buffer cache, so it can grow to fill all of memory.) Since mean file sizes do not seem to be increasing as rapidly [1], perhaps as main memories get larger, using delayed writes would increasingly reduce disk traffic because of write-hits in the cache. (Traces do show that a few large files are getting much larger [1], and so caching algorithms should perhaps switch to write-through for any file larger than a certain size.)

Although increasing DRAM densities lead to larger buffer caches and perhaps more use of delayed writes, disk technology trends are less encouraging. Disk access times have improved by perhaps one-third in ten years. Disk densities are increasing more rapidly, doubling every three years [5]. Disk bandwidths tend to scale as the square root of disk density (since density improvements come from both higher signal rates and smaller track spacings), and also benefit from small increases in rotation rate (from 3600 RPM to 5400 RPM), so over the past decade they have improved by perhaps a factor of six.

This means that the time it takes to write the entire buffer cache to disk is growing, in absolute terms. This is the key problem for the PU update policy: the queueing delays caused by its burst of write requests will get worse in the future.

Delayed writes typically are queued in no particular order. If the disk driver does nothing to schedule the write operations more carefully, the rate at which the queue can be drained depends mostly on the disk's average access time. In table 2-1, I show how long it takes to write the entire buffer cache (assuming this is 10% of main memory) for systems typical of 1983 and 1993, and I rashly project current trends 10 years into the future.

If the disk drive system can optimize the order of writes in the queue, in the best case the queue can be drained at full disk bandwidth. (Many UNIX disk drivers do sort requests to avoid seeks [6], and some

Year	RAM size	Buffer size	Number of buffers	Avg. disk access time	Time to write all buffers
1983	1 MB	512 B	205	35 msec	7.2 sec
1993	64 MB	4 KB	1638	15 msec	26 sec
2003	4 GB	64 KB	6554	6 msec	39 sec

Table 2-1: Scaling for random write of entire buffer cache

Year	RAM size	Buffer cache size	Disk Bandwidth	Time to write all buffers
1983	1 MB	103 KB	1 MB/s	0.1 sec
1993	64 MB	6554 KB	5.5 MB/s	1.2 sec
2003	4 GB	410 MB	30 MB/s	13.6 sec

Table 2-2: Scaling for sequential write of entire buffer cache

modern disk drives themselves reorder requests.) In table 2-2, I show how the delays for this process scale over time. These numbers are much better than those in table 2-1, but they are probably unattainable, and in any case they are also getting worse.

3. Implementation

In this section, I discuss the implementation of various update policies, including the original UNIX implementation of PU, my approximate implementation of IPU, and Sprite's implementation of a similar policy.

3.1. 4.2BSD implementation of the PU policy

Before describing how I implemented the IPU policy, I will describe the ULTRIX implementation of the PU policy. My code is a simple modification of the ULTRIX implementation.

Every 30 seconds, a daemon process (`/etc/update`) wakes up and does a `sync()` system call. This system call schedules writes for certain file system meta-data (the superblock, for example), and then calls the `bflush()` routine to update delayed writes.

In the original 4.2BSD implementation, `bflush()` traversed a queue containing all of the valid blocks in the buffer cache, and scheduled an immediate write for each delayed-write block. Once a block was written and removed from the queue, the algorithm started again from the beginning of the queue; this is done because the queue could be manipulated by another process while the `bflush()` is waiting for the write to complete. Thus, in the worst case this required time proportional almost to the square of the

size of the buffer cache, and as memories grew larger, the /etc/update process started to consume a large fraction of the CPU.

Recent versions of ULTRIX solve this problem by keeping a separate list of delayed-write blocks (i.e., blocks that are dirty and have not yet been queued for the disk). The *bflush()* routine simply traverses this list once; it need not examine clean blocks, nor does it have to examine any block more than once.

3.2. Implementation of the IPU policy

Carson and Setia point out that to implement a pure IPU policy would require a somewhat complex timer mechanism. Since the timers in the UNIX kernel are quantized, if one wants to issue write operations in the same order as the blocks are dirtied, then one also needs to maintain a separate ordered queue. The overhead of the queue and timers may not be onerous, but it does complicate the kernel.

But why implement pure IPU? If a practical implementation must use quantized time, why not use a relative coarse grain? If the algorithm that moves delayed-write blocks onto the disk queues runs, say, once per second, then the queues will receive bursts of writes, but the bursts will (on average) be about 3% as large as they would be with the PU policy and a 30-second update interval. There will also be a 1-second quantization error in the maximum vulnerable period for a dirty block, but who cares?

I chose to implement this kind of approximate IPU (or "AIPU"). I created an alternative version of the *sync()* system call, *smoothsync()*, that takes as its parameter the age at which a dirty block should be written to disk². The *smoothsync()* system call invokes a modified version of the *bflush()* routine, called *bflush_smooth()*. The main difference is that *bflush_smooth()* only schedules a buffer for writing if it has been dirty for longer than the specified threshold.

I replaced the usual /etc/update program, which simply calls *sync()* once every thirty seconds, with one that calls *smoothsync(30)* once a second. This program also forces the file system meta-data to disk once every 30 seconds.

The system must also record the time at which a buffer becomes dirty, using a timestamp field in the

header associated with each buffer. I did this by adding a few lines of code to a routine called *brelse()*, which is the only place where a buffer is placed on the delayed-write list. At this point, if the timestamp field is zero, then it is set to the current time; otherwise, it is left alone. The *brelse()* routine is also the only place where buffers are placed on the list of clean buffers; at this point, the timestamp field is set to zero.

Thus, a clean buffer always has a zero timestamp. A dirty buffer always has a timestamp reflecting when it was first dirtied. Further modifications of a dirty block do not update the timestamp; otherwise, a block that was touched more often than once every 30 seconds would never be written to the disk.

ULTRIX already includes a timestamp field, *busy_time*, in the buffer header. This is used only when a buffer is busy (i.e., on a disk operation queue), and so is never used when a buffer is on the delayed-write list. Therefore, it can be "time-shared" between these two uses. Other operating systems, including 4.xBSD, do not have such a timestamp field, and so the implementation of IPU would require its addition. The space overhead is small; using modular arithmetic, a one-byte field would allow maximum ages under 255 seconds.

This modification re-introduces the possibility of N^2 behavior in the worst case (that is, when about half of the buffer cache is due to be written during a single interval). I solved this by using an extra queue. The *bflush_smooth()* routine starts by traversing the delayed-write queue and moving ready-to-write buffers from there onto a pending-write queue; this can be done without blocking, and in time linear in the size of the buffer cache. In the second phase, *bflush_smooth()* writes the blocks on the pending-write queue. It could block during this phase, but because it simply pulls the first entry off of the pending-write queue, the algorithm is linear in the number of ready-to-write blocks, and so is also linear in the size of the buffer cache, even in the worst case.

3.3. Sprite's implementation of an IPU policy

The Sprite operating system [9] implements an approximate IPU policy, although somewhat different from the one I implemented. Sprite keeps track of the first-dirty time for the oldest dirty block of each file. Every five seconds, it scans all the dirty files in its cache, and if a file's oldest dirty block is more than 30 seconds old, all of the file's dirty blocks are written back [4]. Because this policy can in theory cause write-backs of fairly young blocks, it may perform somewhat differently from IPU or

²Actually, instead of creating a true system call, I added an *ioctl* request, since this involved writing less code. The net effect should be identical. I added one more *ioctl*, to write file system meta-data to disk; this can be called once every 30 seconds, to preserve existing *sync()* semantics.

AIPU. However, since most files are open for only brief periods [1], and so normally all writes to a file happen more or less simultaneously, the average lifetime of a dirty block should be close to 30 seconds.

4. Results

In this section, I describe some simple measurements comparing my implementation of IPU against the original PU policy. All of these measurements were done using a modified ULTRIX version 4.3 kernel, running on either of two DECstation systems; the hardware is summarized in table 4-1. The SCSI disk drives used apparently do not reorder requests, and the ULTRIX version 4.3 SCSI device driver does not sort requests before issuing them.

In all of the experiments in sections 4.1 and 4.2, two processes ran simultaneously:

- A write-load generator, configured to dirty half the blocks in the buffer cache every 30 seconds, at a rate set so that no block would be touched twice in one minute. This means that none of the writes would hit a dirty block in the cache.
- A read-load generator, which read 10,000 randomly chosen blocks from a large file, measured the read-response time for each block, and generated a histogram of the delays.

On the faster system, the buffer cache held 1228 blocks, and the read-load generator used a 34 Mbyte file. On the slower system, the buffer cache held 614 blocks, and the read-load generator used a 32 Mbyte file. Both the generators used files stored on the same disk.

I varied the system configuration, enabling or disabling the use of delayed writes for full data blocks, and changing the update policy. Six trials were done for each configuration. I did not measure a pure write-through configuration, since I do not expect any modern system to use pure WT, given its known poor behavior.

4.1. Local tests

The first set of tests show how the update policy and use of delayed writes affects response time for reads when the disk is local to the generating host. Figures 4-1 and 4-2 show the results for the faster and slower systems, respectively.

The figures show a point for each trial, plotted with mean read response time on the horizontal axis, and the standard deviation of read response time on the vertical axis. Open squares show results for the default configuration (asynchronous writes, PU

policy): moderately high mean response time, but low variance. When I enabled delayed writes without changing the update policy, the mean response time dropped somewhat, but the variance increased tremendously (open circles). I then switched to the AIPU policy (filled circles), which reduced the variance without markedly increasing the mean. One could also use the AIPU policy without delayed writes (filled squares); this results in about the same mean as the default configuration, but slightly less variance.

At first, it seemed strange that the mean read response time is lower for delayed writes, since the write-load generator was constructed to avoid dirty-block cache hits; that is, the total number of write operations should be the same in either case. However, the generator does its writes sequentially, which means that when a group of delayed writes is sent to the disk, they are likely to be directed at nearby disk locations, and many end up in the same cylinder group. This reduces the average number of disk seeks per write (relative to non-delayed writes), and so reduces the load on the disk. Since the read-load generator issues random-access reads as fast as possible, disk seeks are probably the rate-limiting bottleneck, and a reduction in the number of write-related seeks leaves more disk-seek capacity to be used for reads. Also, issuing multiple writes to the same region of the disk may reduce rotational latencies.

For example, figure 4-1 shows that without delayed writes, the system can support about 32 reads/sec., and the write-load generator in this case issues about 20 writes/sec., for a total load of about 52 disk operations/sec. Based on the average access time shown in table 4-1, the disk drive should support about 56 random-access operations/sec, which corresponds closely. With delayed writes, the read rate increases to about 38 reads/sec., which means that the disk should only be doing about 14-17 random writes/sec. This suggests that some fraction of the 20 blocks written are being combined with neighbors.

Figures 4-1 and 4-2 show the standard deviation of the response times, but this hides how truly awful things can be with the PU policy. Figures 4-3 and 4-4 show histograms of response time for the faster and slower systems, respectively. In these histograms, all six trials for each configuration have been combined, and the *x*-axis has been divided into logarithmic buckets.

The danger of combining delayed writes and the PU policy now shows up clearly (open circles on the histograms). Because the PU policy puts all the dirty

Description	CPU type	SPECMark rating	Disk type	Average access time	Bandwidth
Faster system	DECstation 5000 model 200	18.5	RZ58	18 msec	3.8-5.0 Mbyte/sec
Slower system	DECstation 3100	11.3	RZ57	23 msec	2.2 Mbyte/sec

Table 4-1: Systems used for measurements

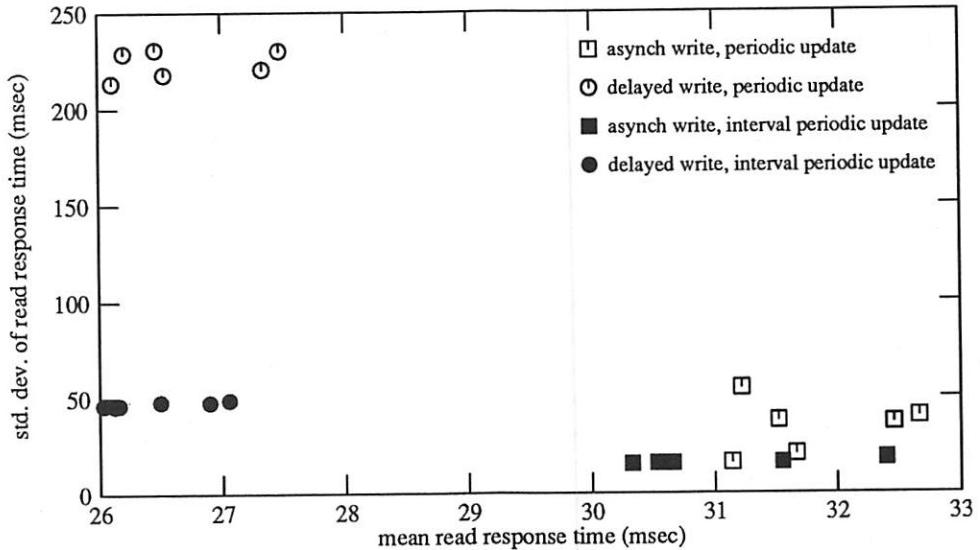


Figure 4-1: Local random reads, fast system

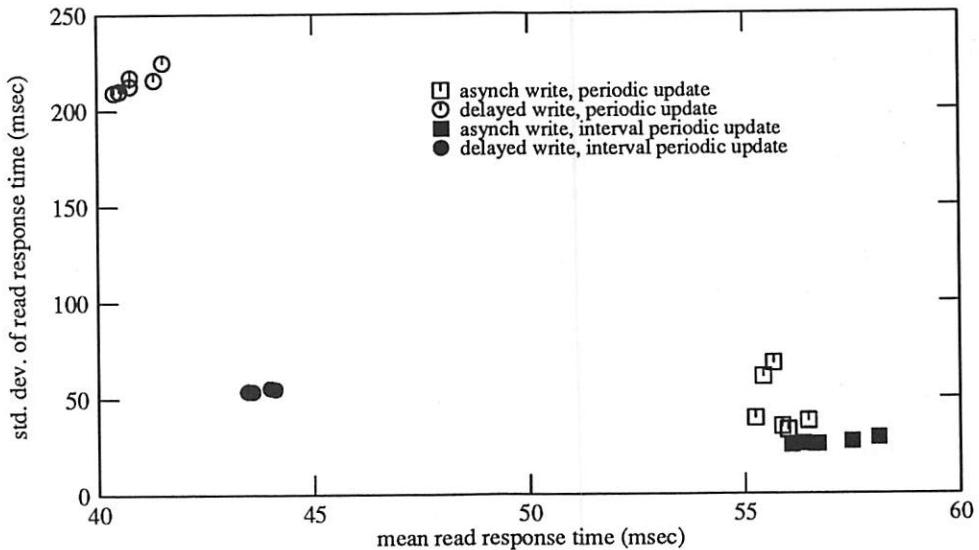


Figure 4-2: Local random reads, slow system

delayed-write blocks on the queue at once, which in these trials could be as high as about 600 blocks, some reads will be delayed by up to 8 or 9 seconds. A significant number are delayed by longer than one second.

With the AIPU policy (filled circles), however, only one thirtieth of the dirty blocks show up on the disk queue at any one time. The graph in figure 4-3

shows a small peak in read-response time at about 300 msec. Since we expect 20 blocks ($600/30$) to be queued once a second, this implies a mean write-delay of about 15 msec, which corresponds closely with the RZ58's specified average access time of 18 msec. More important, this configuration shows a maximum delay of 746 msec, and all but one of the 60,000 samples are below 450 msec. AIPU clearly improves upon PU, in this experiment.

Even when I disabled delayed writes, the PU policy still lead to occasional long delays (up to five or six seconds). In other words, AIPU performs better than PU even if one does not want to abandon the improved safety of using asynchronous writes.

4.2. Remote tests

Most NFS client implementations, to ensure cache consistency and detection of write errors, force delayed writes to the server when a file is closed. This means that NFS clients get little advantage from delayed writes, and do not depend much on the update policy. More recent file service protocols, however, such as Sprite [9], Spritely NFS [15], and NQ NFS [7], use explicit cache-consistency protocols and so can benefit from delayed writes.

I ran a set of experiments using Spritely NFS to access a remote disk, using the “slower” system as the client, and the “faster” system as the server. Both the read-load and write-load generators ran on the same client host. The server host supports PrestoServe™ non-volatile RAM (NVRAM). I ran six trials in each of six configurations, as shown in figure 4-5, using random-access reads. I also ran six trials in each of four configurations, as shown in figure 4-6, using sequential reads; in this set of trials, delayed writes were always used. The sequential-read generator cycled through the blocks of a file much larger than the buffer cache on either the client or server, so no reads were satisfied by the caches.

These experiments showed less conclusive results than the local-disk experiments. For random reads (figure 4-5), delayed write combined with PU results in slightly poorer read response time than delayed write with AIPU, although several AIPU trials exhibited much higher variance than any other trials. PrestoServe also seems to be generally beneficial.

For sequential reads (figure 4-6), AIPU seems mostly to reduce the variation between trials. The mean response time is slightly worse for AIPU than for PU without PrestoServe, and slightly better with PrestoServe.

4.3. Kernel-build benchmark

To see if the update policy had any effect on a “real” application, I measured the time it took to compile and link the entire ULTRIX V4.3 kernel, under different combinations of update policy and write policy. These tests were all run on the “faster” system, with all kernel source and object files on the local disk. This process creates about 43 MB of object files, and a similar amount of temporary file data. I ran three trials in each configuration; the means of the results are shown in table 4-2.

The AIPU policy shows a small but clear advantage over the PU policy, especially when using delayed writes. In fact, when using delayed writes with the PU policy, the net elapsed time is actually slightly worse than the normal ULTRIX configuration. The combination of delayed writes and AIPU is about 2.1% faster than the asynch write/PU combination. Note that this benchmark is rather CPU-bound; on these trials, the CPU idle time averaged between 12% and 14%. I would expect AIPU to show a larger benefit on a more I/O-bound application.

The table also shows the number of disk writes charged to the processes involved in the build. The kernel charges a process the first time a block is dirtied; subsequent writes to a dirty block are not counted. We see that while use of delayed writes substantially decreases the write count, by increasing the chance that a write will hit a dirty block, the AIPU policy provides another big decrease, probably by increasing the average lifetime of a dirty block. The combination of delayed writes and AIPU eliminates over 35% of the disk write operations; AIPU by itself accounts for less than half of the improvement.

The table does not show the number of read I/Os charged, since this hardly varies at all with the update policy, and only a few per cent with the use of delayed writes.

4.4. CPU costs of update mechanism

Does the update policy have any effect on the CPU-time cost of doing the updates? The AIPU policy scans the list of dirty blocks 30 times more often than the PU policy does, so one might expect it to consume more CPU time.

I measured the CPU time charged to the /etc/update update process during the kernel-build benchmark. This includes both user-mode time (which should be nearly zero) and kernel-mode time (which accounts for all CPU time spent in the *bflush()* routine, as well as other activity). It does not include kernel-mode time spent as a result of disk interrupts. The results are shown in table 4-3; note that the underlying measurements were done with 1-second resolution, and so small variations in the results are not significant.

The table shows that, not surprisingly, aggressive use of delayed writes does increase the CPU time spent in finding delayed-write blocks and scheduling them for disk I/O. Contrary to my expectation, however, AIPU actually reduces the CPU cost of doing updates (although the total cost is in either case insignificant).

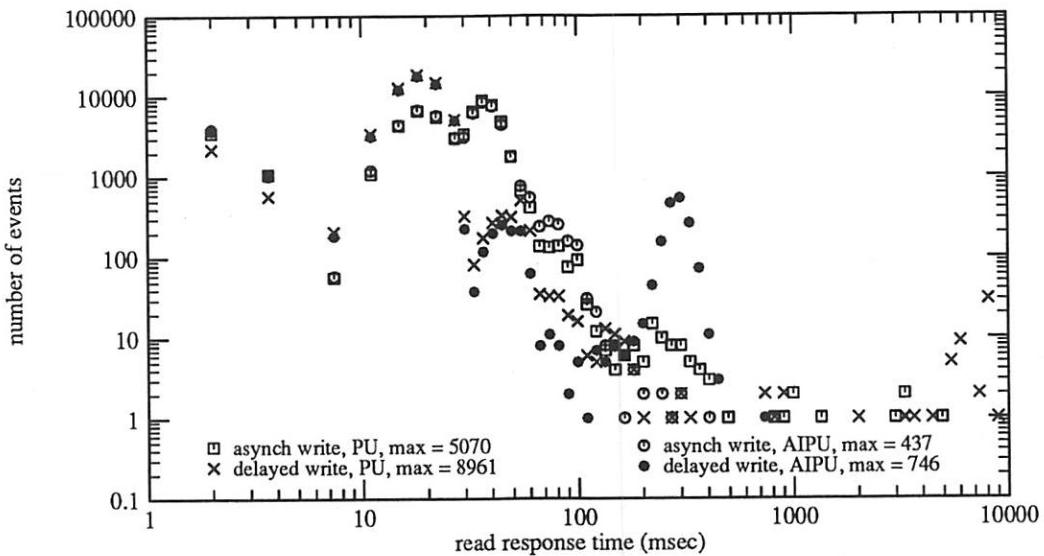


Figure 4-3: Local random reads, fast system (histogram of response times)

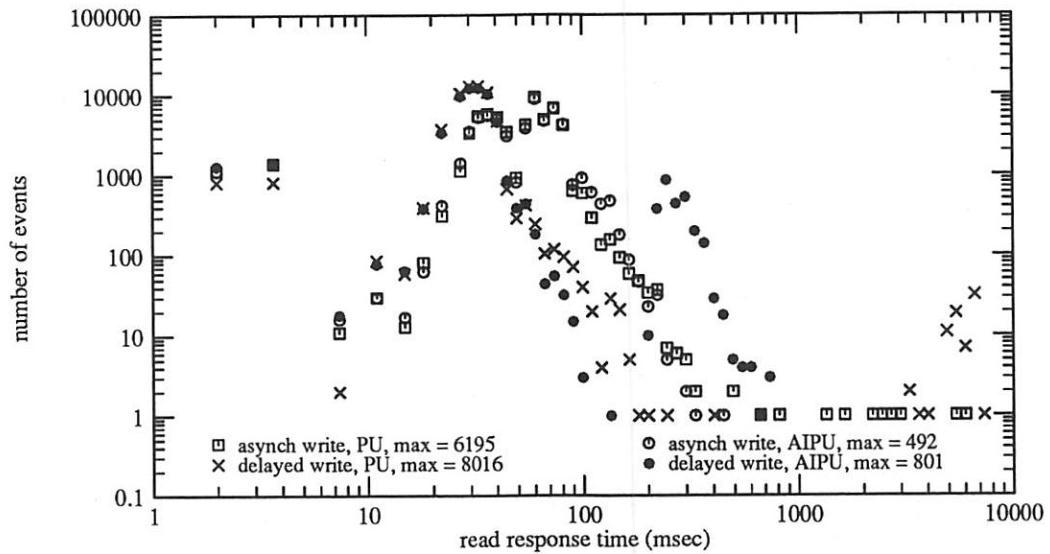


Figure 4-4: Local random reads, slow system (histogram of response times)

I cannot provide a definite explanation, but I suspect that the cause may be the difference in the number of delayed-write blocks actually written to disk. Both algorithms scan every delayed-write block in the buffer cache, and since the AIPU algorithm does this 30 times more often, this suggests that the cost of actually scanning blocks does not dominate the CPU time; the time is probably spent in the device driver³. I ran additional trials using AIPU

with a 30-second interval between updates (but still using a 30-second age threshold for writing back dirty blocks). This took more CPU time than AIPU with a 1-second interval, but still less than PU.

Table 4-3 shows that AIPU scans far more blocks than PU, because PU scans each block exactly once, but AIPU may scan a dirty block many times before deciding that it is old enough to write to disk. However, AIPU actually writes fewer blocks than PU, because AIPU allows the average dirty block to stay in the cache longer. Recall that with PU, the average age of a block when written to disk is 15 seconds, but with AIPU the average is 30 seconds, assuming a uniform rate of file writes. (AIPU-30, with a 30-second threshold and a 30-second interval between updates, yields an average age of 45

³Note that for the kernel-build benchmark, PU with delayed writes scans 16749 blocks in 6.0 CPU seconds, placing a lower bound of $6/16.7 = 0.36$ msec per block scanned. This corresponds to several hundred instruction executions, much more than could be accounted for by the scanning loop itself, so most of this time must be spent in the disk driver.

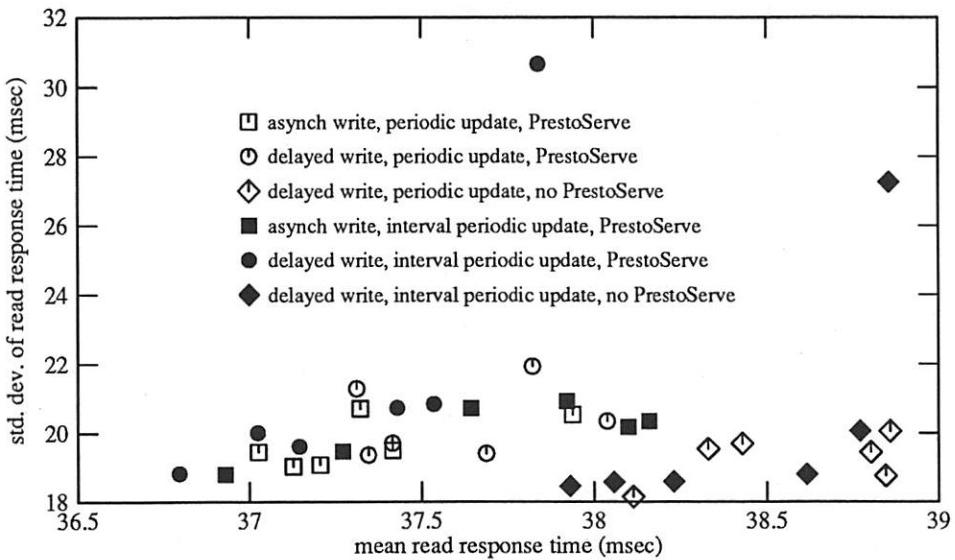


Figure 4-5: Remote random reads

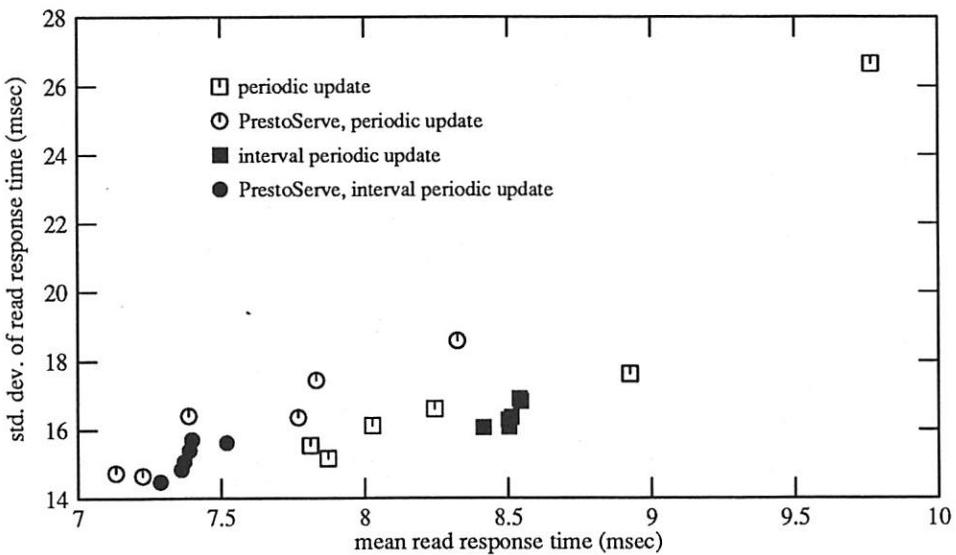


Figure 4-6: Remote sequential reads

	Periodic update	Approx. interval periodic update	Relative elapsed time
Asynch writes	3280 sec. elapsed 40548 writes	3234 sec. elapsed 37854 writes	0.986
Delayed writes	3298 sec. elapsed 31523 writes	3206 sec. elapsed 25966 writes	0.972

Mean values for 3 trials

Table 4-2: Elapsed time on kernel-build benchmark

seconds, and so does slightly fewer writes than AIPU-1).

A dirty block that stays in the cache for a longer time is more likely to be modified again before being written to disk, which results in fewer modifications

of clean blocks, and hence fewer writes to disk. I counted the number of modifications of currently dirty blocks (in the *brelse()* routine); the numbers are shown in table 4-3. AIPU with delayed writes results in moderately more dirty-block modifications (426K vs. 419K for PU); the difference in dirty-block

	PU, async writes	PU, delayed writes	AIPU-1, async writes	AIPU-1, delayed writes	AIPU-30, async writes	AIPU-30, delayed writes
CPU time used by /etc/update (mean of 3 trials)	4.3 sec.	6.0 sec.	2.0 sec.	2.3 sec.	2.7 sec.	2.7 sec.
Blocks scanned by /etc/update	9531	16749	290214	508448	14593	25847
Blocks written by /etc/update	9531	16749	5375	10023	5266	9519
Dirty blocks modified during benchmark	391K	419K	393K	426K	405K	426K
<i>biowait()</i> sleep events during benchmark	8234	8274	7607	8121	7496	7751

Table 4-3: Statistics for kernel-build benchmark

modifications is roughly the same as the difference in delayed-write disk I/Os.

The reduction in actual disk I/Os, caused by longer cache lifetimes and more dirty-block modifications, may account for all of the elapsed-time advantage of AIPU over PU on the kernel-build benchmark. As table 4-3, with AIPU-1 and especially AIPU-30, the kernel “sleeps” less often for disk I/O than it does with PU (although I could not measure the total sleep time). However, this effect should not contribute to the random-access results in sections 4.1 and 4.2, since these experiments were constructed to avoid any cache hits on file writes.

4.5. Burstiness of file writes

The advantage of AIPU over PU is that the latter clumps together all delayed writes from a 30-second period into a single burst of writes, while the former preserves the original spacing of the file writes (with 1-second resolution). If the file writes themselves arrive in bursts, this eliminates AIPU’s advantage. In the worst case, when all file writes during a 30-second period occur nearly at once, both policies should perform the same.

In the experiments reported in sections 4.1 and 4.2, the write-load generator distributed file writes uniformly over time, which is the best case for AIPU. The kernel-build benchmark should be more representative of real use; how bursty is its file-write pattern? (Note that this kind of single-user benchmark is more likely to exhibit burstiness than a multi-user benchmark, because the latter will tend to spread out the file-system load among several jobs.)

I modified the *bflush()* and *bflush_smooth()* routines to keep a histogram of the number of blocks they queue for the disk on each invocation. I then ran one trial of the kernel-build benchmark for each of

PU, AIPU-1, and AIPU-30. In all cases, the update period (age at which a block is queued to the disk) was 30 seconds.

With AIPU-1, since blocks are queued 30 seconds after the corresponding file write, the burstiness in queue-batch size directly mirrors the burstiness in the file-write pattern (with 1-second granularity).

The results are shown in figure 4-7, in the form of cumulative distributions for the number of blocks written as a function of the burst size. Each policy writes a different number of blocks (see table 4-3), so the curves do not end at the same ordinate. For AIPU-1 (with 1 second between updates and a 30-second period), 95% of all delayed-write blocks written were queued by *bflush_smooth()* in bursts of 40 blocks or fewer. The largest burst contained 104 blocks. In other words, the application delivered relatively small bursts of writes to the file system.

For PU (with 30 seconds between updates), 95% of the delayed-write blocks queued were in bursts of more than 62 blocks, and 50% were queued in bursts of more than 182 blocks. The largest burst contained 344 blocks, which (assuming an 18 msec. mean access time) delayed any subsequent synchronous operation by over 6 seconds.

To summarize figure 4-7, the kernel-build benchmark does indeed spread out its writes over periods longer than a second. This results in far smaller disk-queue bursts when AIPU-1 is used than when PU is used.

5. Future work

Carson and Setia proposed using a periodic update with read priority (PURP) policy. Although I resisted implementing PURP, because of its greater complexity, in the general case one cannot avoid long disk queues even with IPU or an approximation. For

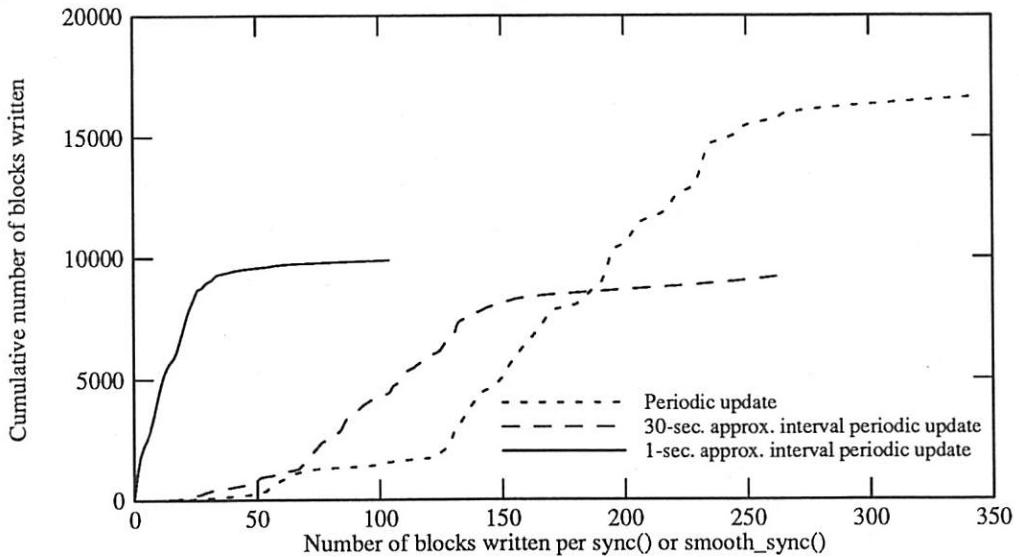


Figure 4-7: Burst-size distributions for writes during kernel-build benchmark

example, if an application manages to dirty the entire buffer cache within a second or so, thirty seconds later an IPU policy will schedule writes for all those blocks, and the effect will be the same as with the PU policy. In other words, IPU depends on a relatively uniform distribution of file writes (across time) to achieve its more uniform distribution of disk writes.

If long disk queues are inevitable, and most of the entries on such queues are inherently asynchronous, then giving priority to reads and synchronous writes should improve response time. I suspect that, even with a priority scheme, an IPU policy could outperform PU. Suppose the buffer cache is entirely filled by dirty blocks; then, a read operation must wait until the system cleans a block before it can complete. The IPU policy generates clean blocks once a second or so (assuming a uniform distribution of disk writes, across time), but the PU policy only does this every 30 seconds. Thus, with PU, reads would be more likely to block waiting for a free buffer. This is speculation; we need experiments, simulation, or more formal analysis to discover the truth.

Peacock [11, 12] has described several systems that use PURP, but apparently did not explore modified update policies. He found that adding read priority to System V Release 4, in which the buffer cache can be quite large, actually reduced benchmark performance by preventing asynchronous requests from getting a sufficient share of the disk.

My experiments have all used a 30-second period for PU and for the lifetime of dirty blocks in IPU, and a one-second granularity for IPU. I suspect that use of different periods and granularities, within reason-

able limits, will not make a big difference, but this should be the subject of additional experiments.

Some UNIX file system implementations attempt to cluster several blocks together when performing a disk write [8, 11]. That is, if the cache contains several dirty blocks that are adjacent on disk, the file system or disk driver attempts to write them all at once, which improves throughput by eliminating seeks and rotational delays. Clustering can be done in several different ways, and may interact with the delayed write policy (that is, are all data writes delayed, or only partially-filled blocks?) and with the update policy. The ULTRIX systems tested in section 4 use a clustering algorithm; I have not done experiments to see if this affects the relative performance of update policies.

6. Summary and conclusions

- The experiments described in this paper show that
 - Use of delayed writes can improve overall file system performance, including read response times, on both synthetic and actual workloads,
 - But when delayed writes are combined with the traditional periodic update policy, variance in read response time increases significantly, and benchmark performance may decrease,
 - So one should use a better update policy, such as interval periodic update or an approximation, whenever one uses a delayed write policy.

I also showed that one can easily implement an approximate interval periodic update policy, with

remarkably limited changes to the kernel of a traditional operating system.

Acknowledgements

Scott Carson and Sanjeev Setia helped to encourage me to perform these experiments. John Ousterhout, Jim Gray, Kent Peacock, and the anonymous reviewers provided useful comments during the preparation of this paper.

References

- [1] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a Distributed File System. In *Proc. 13th Symposium on Operating Systems Principles*, pages 198-212. Pacific Grove, CA, October, 1991.
- [2] Scott D. Carson and Sanjeev Setia. Analysis of the Periodic Update Write Policy For Disk Cache. *IEEE Transactions on Software Engineering* 18(1):44-54, January, 1992.
- [3] Anna Hac. Design Algorithms for Asynchronous Write Operations in Disk-Buffer-Cache Memory. *J. Systems Software* 16(3):243-253, November, 1991.
- [4] John Hartman. Private communication. 1993.
- [5] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, 1990.
- [6] Samuel J. Leffler, Marshall Kirk McCusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, Reading, MA, 1989.
- [7] Rick Macklem. Not Quite NFS, Soft Cache Consistency for NFS. In *Proc. Winter 1994 USENIX Conference*, pages 261-278. San Francisco, CA, January, 1994.
- [8] L. W. McVoy and S. R. Kleiman. Extent-like Performance from a UNIX File System. In *Proc. Winter 1991 USENIX Conference*, pages 33-43. Dallas, TX, January, 1991.
- [9] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems* 6(1):134-154, February, 1988.
- [10] John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proc. 10th Symposium on Operating Systems Principles*, pages 15-24. Orcas Island, WA, December, 1985.
- [11] J. Kent Peacock. The Counterpoint Fast File System. In *Proc. Winter 1988 USENIX Conference*, pages 243-249. Dallas, TX, February, 1988.
- [12] J. Kent Peacock. File System Multithreading in System V Release 4 MP. In *Proc. Winter 1992 USENIX Conference*, pages 19-29. San Antonio, TX, June, 1992.
- [13] Dennis M. Ritchie. Private communication. 1994.
- [14] D.M. Ritchie and K. Thompson. The UNIX Time-Sharing System. *AT&T Technical Journal* 57(6):1905-1929, July-August, 1978.
- [15] V. Srinivasan and Jeffrey C. Mogul. Spritely NFS: Experiments with Cache-Consistency Protocols. In *Proc. 12th Symposium on Operating Systems Principles*, pages 45-57. Litchfield Park, AZ, December, 1989.

UNIX is a registered trademark of X/Open Company, Ltd. ULTRIX and DECstation are trademarks of Digital Equipment Corporation. PrestoServe is a trademark of Legato, Inc.

Jeffrey Mogul received an S.B. from the Massachusetts Institute of Technology in 1979, and his M.S. and Ph.D. degrees from Stanford University in 1980 and 1986. Since 1986, he has been a researcher at the Digital Equipment Corporation Western Research Laboratory, working on network and operating systems issues for high-performance computer systems. He is a member of ACM, Sigma Xi, ISOC, and CPSR, the author or co-author of several Internet Standards, an associate editor of *Internetworking: Research and Experience*, and was Program Committee Chair for the Winter 1994 USENIX Technical Conference.

Address for correspondence: Digital Equipment Corporation Western Research Laboratory, 250 University Avenue, Palo Alto, California, 94301 (mogul@wrl.dec.com)

The Desktop File System

Morgan Clark

Stephen Rago

Programmed Logic Corporation

200 Cottontail Lane

Somerset, NJ 08873

Abstract

This paper describes the structure and performance characteristics of a commercial file system designed for use on desktop, laptop, and notebook computers running the UNIX operating system. Such systems are characterized by their small disk drives dictated by system size and power requirements. In addition, these systems are often used by people who have little or no experience administering Unix systems. The Desktop File System attempts to improve overall system usability by transparently compressing files, increasing file system reliability, and simplifying administrative interfaces. The Desktop File System has been in production use for over a year, and will be included in future versions of the SCO Open Desktop Unix system. Although originally intended for a desktop environment, the file system is also being used on many larger, server-style machines.

1. Overview

This paper describes a commercial file system designed for use on desktop, laptop, and notebook computers running the UNIX operating system. We describe design choices made and discuss some of the interesting ramifications of those choices. The most notable characteristic of the file system is its ability to compress and decompress files "on-the-fly." We provide performance information that proves such a file system is a viable option in the Unix marketplace.

When we use the term "commercial file system," we mean to imply two things. First, the file system is used in real life. It is not a prototype, nor is it a research project. Second, our design choices were limited to the scope of the file system. We were not free to rewrite portions of the base operating system to meet our needs, with one exception (we provided our own routines to access the system buffer cache).

1.1 Goals

Our goals in designing the Desktop File System (DTFS) were influenced by our impressions of what the environment was like for small computer systems, such as desktop and laptop computers. The physical size of these systems limits the size of the power supplies and hard disk drives that they can use at a reasonable cost. Systems that are powered by batteries attempt to use small disks to minimize the power drained by the disk, thus increasing the amount of time that the system can be used before requiring that the batteries be recharged.

It is common to find disk sizes in the range of 80 to 300 Megabytes in current 80x86-based laptop and notebook systems. Documentation for current versions of UnixWare recommend a minimum of 80 MB of disk space for the personal edition, and 120 MB for the application server. Similarly, Solaris documentation stipulates a minimum of 200 MB of disk space. These recommendations do not include space for additional software packages.

We also had the impression that desktop and notebook computers were less likely to be administered properly than larger systems in a general computing facility, because the primary user of the systems will probably be performing the administrative procedures, often without the experience of professional system administrators. These impressions led us to the following goals:

- Reduce the amount of disk space needed by conventional file systems.
- Increase file system robustness in the presence of abnormal system failures.
- Minimize any performance degradation that might arise because of data compression.
- Simplify administrative interfaces when possible.

The most obvious way to decrease the amount of disk space used was to compress user data. (We

use the term “user data” to refer to the data read from and written to a file, and we use the term “meta data” to refer to accounting information used internally by the file system to represent files.) Our efforts did not stop there, however. We designed the file system to allocate disk inodes as they are needed, so that no space is wasted by unused inodes. In addition, we use a variable block size for user data. This minimizes the amount of space wasted by partially-filled disk blocks.

Our intent in increasing the robustness of the file system stemmed from our belief that users of other desktop systems (such as MS-DOS) would routinely shut their systems down merely by powering off the computer, instead of using some more gradual method. As it turned out, our eventual choice of file structure required us to build robustness in anyway.

From the outset, we realized that any file system that added another level of data processing would probably be slower than other file systems. Nevertheless, we believed that this would not be noticeable on most systems because of the disparity between CPU and disk I/O speeds. In fact, current trends indicate that this disparity is widening as CPU speeds increase at a faster pace than disk I/O speeds [KAR94]. Most systems today are bottlenecked in the I/O subsystem [OUS90], so the spare CPU cycles would be better spent compressing and decompressing data.

Our assumptions about typical users led us to believe that users would not know the number of inodes that they needed at the time they made a file system, and some might not even know the size of a given disk partition. We therefore endeavored to make the interfaces to the file system administrative commands as simple as possible.

1.2 Related Work

Several attempts to integrate file compression and decompression into the file system have been made in the past. [TAU91] describes compression as applied to executables on a RISC-based computer with a large page size and small disks. While compression is performed by a user-level program, the kernel is responsible for decompression when it faults a page in from disk. Each memory page is compressed independently, with each compressed page stored on disk starting with a new disk block. This simplifies the process of creating an uncompressed in-core image of a page from its compressed disk image. DTFS borrows this technique.

The disadvantage with this work is that it doesn't fully integrate compression into the file system. Only binary executable files are compressed,

and applications that read the executables see compressed data. These files must be uncompressed before becoming intelligible to programs like debuggers, for example. The compression and decompression steps are not transparent.

In [CAT91], compression and decompression are integrated into a file system. Files are compressed and decompressed in their entirety, with the disk split into two sections: one area contains compressed file images, and the other area is used as a cache for the uncompressed file images. The authors consider it prohibitively expensive to decompress all files on every access, hence the cache. This reduces the disk space that would have been available had the entire disk been used to hold only compressed images.

The file system was prototyped as an NFS server. Files are decompressed when they are first accessed, thus migrate from the compressed portion of the disk to the uncompressed portion. A daemon runs at off-peak hours to compress the files least-recently used, and move them back to the compressed portion of the disk.

Transparent data compression is much more common with the MS-DOS operating system. Products like Stacker have existed for many years. They are implemented as pseudo-disk drivers, intercepting I/O requests, and applying them to a compressed image of the disk [HAL94].

We chose not to implement our solution as a pseudo-driver because we felt that a file system implementation would integrate better into the Unix operating system. For example, file systems typically maintain counters that track the number of available disk blocks in a given partition. A pseudo-driver would either have to guess how many compressed blocks would fit in its disk partition or continually try to fool the file system as to the number of available disk blocks. No means of feedback is available, short of the pseudo-driver modifying the file system's data structures. If the pseudo-driver were to make a guess at the time a file system is created, then things would get sticky if files didn't compress as well as expected.

In addition, a file system implementation gave us the opportunity to employ transaction processing techniques to increase the robustness of the file system in the presence of abnormal system failures. A pseudo-driver would have no knowledge of a given file system's data structures, and thus would have no way of associating a disk block address with any other related disk blocks. A file system, on the other hand, could employ its knowledge about the interrelationships of disk blocks to ensure that files are always kept in a consistent state.

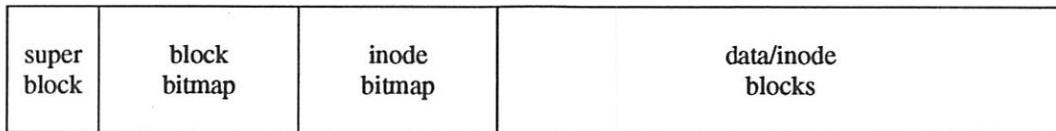


Figure 1. File System Layout

Much more work has gone into increasing file system robustness than has gone into integrating compression into file systems. Episode [CHU92], WAFL [HIT94], and the Log-structured File System [SEL92] all use copy-on-write techniques, also known as shadow paging, to keep files consistent. Copies of modified disk blocks are not associated with files until the associated transaction is committed. (Shadow paging is discussed in Section 5.1.)

2. File System Layout

Figure 1 shows the file system layout of DTFS. The super block contains global information about the file system, such as the size, number of free blocks, file system state, etc. The block bitmap records the status (allocated or free) of each 512-byte disk block. Similarly, the inode bitmap, which is the same size as the block bitmap, identifies the disk blocks that are used for inodes. Finally, the rest of the disk partition is available for use as user data and meta data.

The only critical information kept in the super block is the size of the file system. The rest of the information can be reconstructed by `fsck`. Similarly, the block bitmap can be reconstructed by `fsck`, should it be corrupted by a bad disk block. The inode bitmap is more critical, however, as `fsck` uses it as a guide to identify where the inodes are stored in the file system (inode placement is discussed in the next section). If the inode bitmap is corrupted, files will be lost, so users ultimately have to rely on backups to restore their files.

2.1 Inode Placement

As stated previously, one way DTFS saves space is by not preallocating inodes. Any disk block is fair game to be allocated as an inode. This has several interesting repercussions.

First, there is no need for an administrator to guess how many inodes are needed when making a file system. The number of inodes is only limited by the physical size of the disk partition and the size of the identifiers used to represent inode numbers. This simplifies the interface to `mkfs`.

Second, the inode allocation policy renders the NFS generation count mechanism [SAN85] ineffective. In short, when a file is removed and its link

count goes to 0, an integer, called the *generation count*, in the disk inode is incremented. This generation count is part of the file handle used to represent the file on client machines. Thus, when a file is removed, active handles for the file are made invalid, because newly-generated file handles for the inode will no longer match those held by clients.

With DTFS, the inode block can be reallocated as user data, freed, and then reallocated as an inode again. Thus, generation counts cannot be retained on disk. We solve this problem by replacing the generation count with the file creation time concatenated with a per-file-system rotor that is incremented every time an inode is allocated.

2.2 File Structure

A DTFS file is stored as a B⁺tree [COM79], with the inode acting as the root of the tree. Two factors led us to this design. The first of these is the need to convert between logical file offsets and physical data offsets in a file. For example, if an application were to open a 100000-byte and seek to byte position 73921, we would need an efficient way to translate this to the disk block containing that data. Since the data are compressed, we cannot simply divide by the logical block size to determine the logical block number of the disk block containing the data.

A simple solution would be to start at the beginning of the file and decompress the data until the requested offset is reached. This, however, would waste too much time. To make the search more efficient, we use logical file offsets as the keys in the B⁺tree nodes. The search is bounded by the height of the tree, so it is doesn't cost any more to find byte $n+100000$ than it costs to find byte n .

The second factor is a phenomenon we refer to as *spillover*. Consider what happens when a file is overwritten. The new data might not compress as well as the existing data. In this case, we might have to allocate new disk blocks and insert them into the tree. With the B⁺tree structure, we can provide an efficient implementation for performing insertions and deletions.

When a file is first created, it consists of only an inode. Data written to the file are compressed, stored in disk blocks, and the disk block addresses

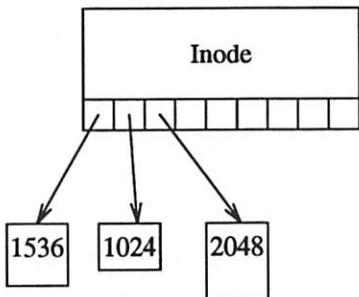


Figure 2. A One-level File

and lengths are stored in the inode (see Figure 2). DTFS uses physical addresses instead of logical ones to refer to disk blocks, because a “block” in DTFS can be any size from 512 bytes to 4096 bytes, in multiples of 512 bytes. In other words, a disk block is identified by its starting sector number relative to the beginning of the disk partition.

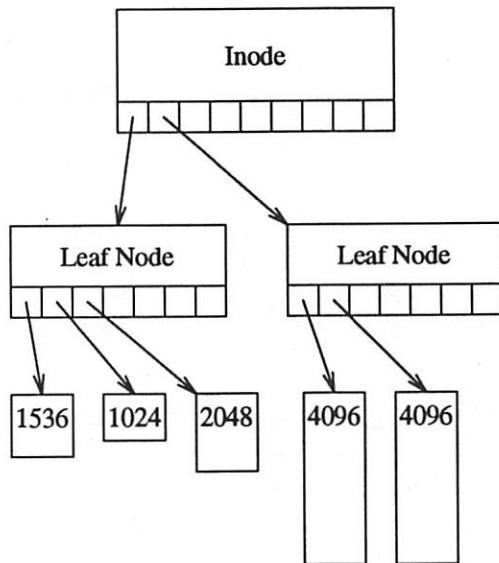


Figure 3. A Two-level File

When the inode’s disk address slots are all used, the next block to be added to the file will cause the B⁺tree to grow one level. (See Figure 3; note that most links between nodes are omitted for clarity). Two disk blocks are allocated to be used as leaf nodes for the tree. The data blocks are divided between the two leaf nodes, and the inode is modified to refer to the two leaf nodes instead of the data blocks. In this way, as the tree grows, all data blocks are kept at the leaves of the B⁺tree.

Figure 4 shows what would happen if the B⁺tree were to grow an additional level. The inode

now refers to interior nodes of the B⁺tree. Interior nodes either refer to leaf nodes or to other interior nodes. Each entry in an interior node refers to a subtree of the file. The key in each entry is the maximum file offset represented by the subtree. The keys are used to navigate the B⁺tree when searching for a page at a particular offset. The search time is bounded by the height of the B⁺tree.

3. Compression

DTFS is implemented within the Vnodes architecture [KLE86]. DTFS compresses the data stored in regular files to reduce disk space requirements (directories remain uncompressed). Compression is performed a page at a time, and is triggered during the vnode putpage operation. Similarly, decompression occurs during the getpage operation. Thus, compression and decompression occur “on-the-fly.”

The choice of compressing individual pages limits the overall effectiveness of adaptive compression algorithms that require a lot of data before their tables are built. Nonetheless, some algorithms do quite well with only a page of data. This was the natural design choice for DTFS, since it was originally designed for UNIX System V Release 4 (SVR4), an operating system whose fundamental virtual memory abstraction is the page. (On an Intel 80x86 processor, SVR4 uses a 4KB page size.)

Each page of compressed data is represented on disk by a *disk block descriptor* (DBD). The DBD contains information such as the logical file offset represented by the data, the amount of compressed data stored in the disk block, and the amount of uncompressed data represented. The DBDs exist only in the leaf nodes of the B⁺tree.

Because each page is compressed individually, DTFS is best suited to decompression algorithms that build their translation tables from the compressed data, thus requiring no additional on-disk storage. The class of algorithms originated by Lempel and Ziv [ZIV77], [ZIV78] are typical of such algorithms.

The original version of DTFS only supported two compression algorithms (an LZW derivative [WEL84] and “no compression”). The latest version of DTFS allows for multiple compression algorithms to be used at the same time. We designed an application programming interface so that customers can add their own algorithms if they should so desire.

4. Block Allocation

With the exception of the super block and bitmaps (recall Figure 1), every disk block in a DTFS file system can be allocated for use as an inode, other meta data, or user data. The basic allocation mechanism is

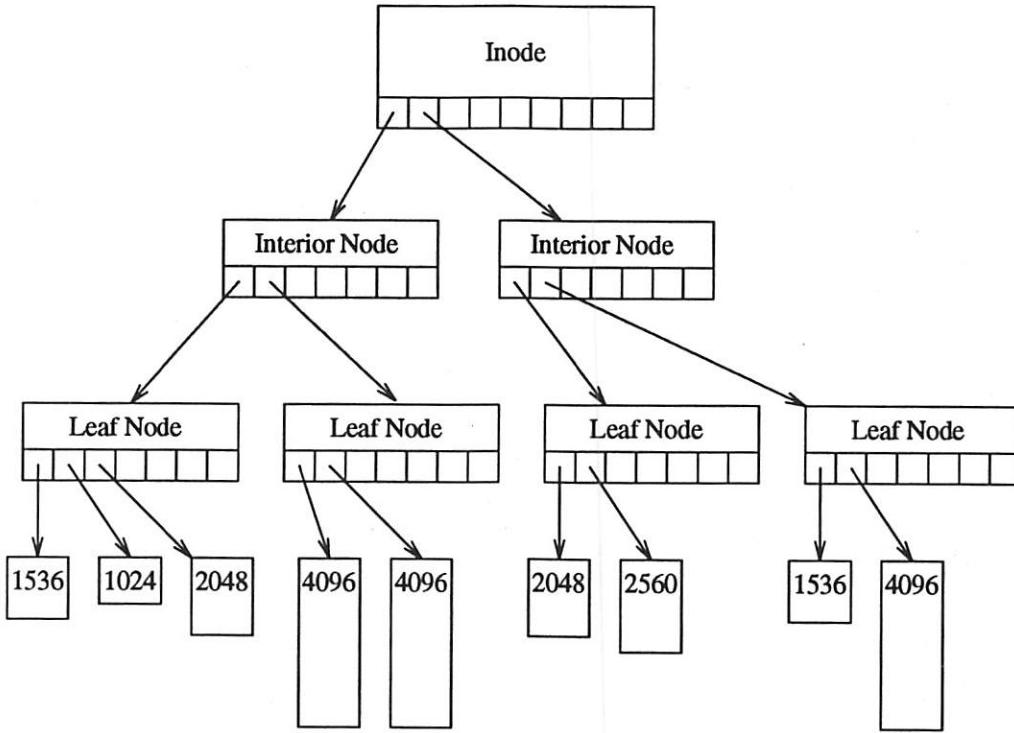


Figure 4. A Three-level File

simple — the file system keeps track of the first available block in the partition and allocates blocks starting from there. Requests to allocate multiple blocks search the file system for a free contiguous region at least as large as the number of blocks requested. If no region is found to be large enough, allocation switches to a first-fit algorithm that returns as many blocks as are contiguously available, starting with the first free block.

Unmodified, this allocation mechanism would cause severe file system fragmentation for several reasons. First, disk blocks backing a page must be allocated at the time a file is written, so that errors like ENOSPC can be returned to the application. Usually pages are written to disk sometime later, as the result of page reuse or file system hardening. If DTFS were to defer block allocation until it wrote a page to disk, then DTFS would be unable to inform the application of a failure to allocate a disk block.

With dynamic data compression, after each page is compressed, the leftover blocks are freed back to the file system. So for every eight-block page there might be a run of, say, four blocks allocated followed by four blocks freed. As each page requires eight disk blocks before compression, no other page would use any of the four-free-block regions until the entire file system is fragmented and allocation switches to first-fit.

The second contributor to fragmentation is the shadow paging algorithm, which requires that every overwritten page have new blocks allocated before the old blocks are freed. The new blocks would be allocated from some free region, most likely in a different place in the file system.

The final reason for fragmentation is that any block can be used for any purpose, so inode blocks and user data blocks could be intermingled. This is partly mitigated by the fact that inode and meta data blocks can use the few-block regions between compressed user data blocks.

We use several techniques to avoid fragmentation. First, user data blocks are allocated in clusters, one cluster per inode. When the DTFS write function attempts to allocate disk space for a user data page, a much larger chunk of space (typically 32 blocks) is actually allocated from the general disk free space pool, and the blocks for the page in question are allocated from this cluster. The remainder of the cluster is saved in the in-core inode for use by later allocations. When the inode is no longer in use, the unused blocks in the cluster are returned to the file system. This has the effect of localizing disk space belonging to a particular file.

Second, after compression, user data blocks are reassigned from within the cluster to eliminate any local fragmentation. For example, if the first page of

a file allocated blocks 20–27 and the second page allocated blocks 28–35, and after compression each page needed the first four blocks of their respective eight-block region, the second page would be reassigned to use blocks 24–27, eliminating the four-block fragment of free space.

Finally, we keep two first-free block indicators, and allocate inode blocks starting from one point and user data blocks from another.

5. Reliability

Our goal of increasing file system reliability was originally based on our belief that naive users might not shut a system down properly. File systems like S5 (the traditional System V file system [BAC86]) and UFS (The System V equivalent of the Berkeley Fast File System [MCK84]) can leave a file with corrupted data in it if the system halts abnormally. Unless an application is using synchronous I/O, when an application writes to a file such that a new disk block needs to be allocated, file systems usually write both the user data and the meta data in a delayed manner, caching them in mainstore and writing them out to disk later, to improve performance.

The problem with this approach is that an abnormal system halt can leave a file containing garbage. Consider what happens when the inode's meta data are written to disk, but the system halts abnormally before the user data are written. (This case is more likely to occur than one might think — the system tries to flush “dirty” pages and buffers to disk over a tunable time period, usually around 60 seconds. Thus, in the worst case, it is possible for 60 seconds to elapse before user data are flushed to disk.) In this case, the inode's meta data will reflect that a newly allocated disk block contains user data, but the actual contents of the disk block have not been written yet, so the file will end up containing random user data. The contents of the disk block can be as innocuous as a block of zeros, or as harmful as a block that came from another user's deleted file, thus presenting a security hole. (When a block is freed or reused for user data, its contents are usually not cleared.)

This reliability problem can be solved by carefully ordering the write operations. If the user data are forced out to disk before the inode's meta data, then the file will never refer to uninitialized disk blocks. The ordering must be implemented carefully, because forcing the user data out to disk before the meta data can hurt performance. On the other hand, delaying the update of the meta data until the user data have found their way to disk can cause changes to the inode to be lost entirely if the system crashes.

Our choice of compressing user data, as it turned out, forced us to increase reliability to keep the user data and meta data in sync. Because a disk block can contain a variable amount of data, and because that data, once compressed, can represent an amount of data greater than the size of the disk block, we require that the meta data and the user data be in sync at all times. The meta data describe the compression characteristics, and if the information were to be faulty, the decompression algorithm might react in unpredictable ways, possibly leading to data corruption or a system panic.

5.1 Shadow Paging

To solve the problem of keeping a file's meta data and user data in sync, we decided to use *shadow paging* [EPP90], [GRA81], with the intent of providing users with a consistent view of their files at all times.

Shadow paging is a technique that can be used to provide atomicity in transaction processing systems. A *transaction* is defined to be a unit of work with the following ACID characteristics [GRA93]: Atomicity, Consistency, Isolation, and Durability.

The atomicity property guarantees that transactions either successfully complete or have no effect. When a transaction is interrupted because of a failure, any partial work done up to that point is undone, causing the state to appear as if the transaction had never occurred. The consistency property ensures that the transaction results in a valid state transition.

The isolation property (also called serializability) guarantees that concurrent transactions do not see inconsistencies resulting from the possibly multiple steps that make up a single transaction. (A single transaction is usually made up of multiple steps.) DTFS uses inode locks to provide isolation.

The durability property (also known as permanence) guarantees that changes made by committed transactions are not lost because of hardware errors. This is usually implemented through techniques, such as disk mirroring, that employ redundant hardware. DTFS does not provide permanence.

Shadow paging provides only the atomicity property. It works in the following way: when a page is about to be modified, extra blocks are allocated to shadow it. When the page is written out, it is actually written to the shadow blocks, leaving the original blocks unmodified. File meta data are modified in a similar manner, except that the node and its shadow share the same disk block.

When an inode is updated, the user data are flushed to disk. Then the modified meta data are written, followed by a synchronous write of the inode (the root of the B⁺tree). Finally, all the original data

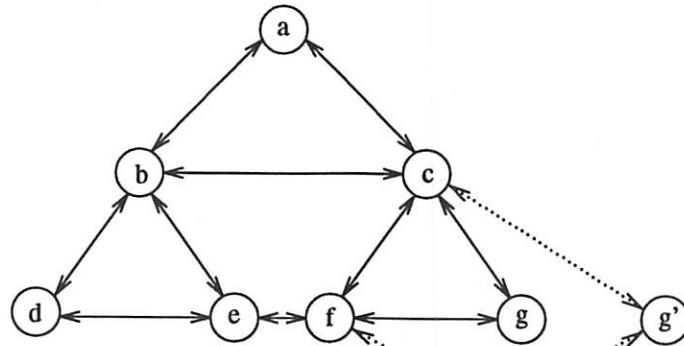


Figure 5. Partial B⁺tree

blocks are freed. If an abnormal system halt occurs before the inode is written to disk, then the file appears as it was at the time of the previous update.

The shadow of a node is stored in the same disk block as the original node is to avoid a ripple effect when modifying part of a file. In Figure 5, node g is shadowed by node g' . When node g' is added to the tree, we would have to allocate new nodes, c' and f' , when we updated the nodes with the new pointer to g' . Then we would have to update all the nodes that have pointers to c and f , and so on, until the entire tree has been updated.

We avoid all this extra work by design: g and g' reside in the same disk block (see Figure 6). This means that when we decide to use g' instead of g , the nodes that point to g don't have to be updated. The way we determine whether to use g or g' is by storing a timestamp in each node to describe which is the most recently modified of the pair.

Every time we write a node to disk, the inode's timestamp is incremented and copied to the node's timestamp field. Before the inode is written to disk, its timestamp is incremented. Any nodes with a timestamp greater than that of the inode were written to disk without the inode being updated, so they are ignored. Otherwise, we choose the node with the largest timestamp of the two. In the event of a system failure, `fsck` will rebuild the block bitmap, and the user data duplicated by shadowing will be freed.

5.2 Quiescence, Sync-on-Close, and the Update Daemon

To increase reliability, we implemented a kernel process, called the *update daemon*, that checkpoints every writable DTFS file system once per second. The daemon performs a sync operation on each file system, with the enhancement that if the sync completes successfully without any other process writing

to the file system during that time, the file system state is set to "clean" and the super block is written out to disk.

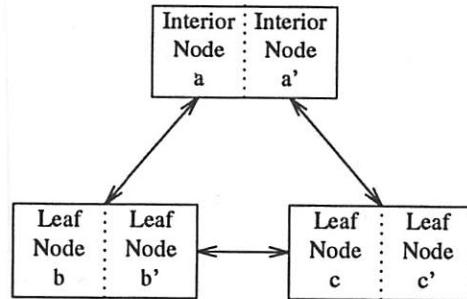


Figure 6. Meta Data Shadows

From this point until the next time any process writes to the file system, a system crash will not require that `fsck` be run on that file system. Studies of our main development machine, with DTFS on all user data partitions, showed that the file systems were in the clean state more than 98% of the time. (DTFS supports an `ioctl` that reports the file system state.)

Another reliability enhancement was to write every file to disk synchronously, once the last reference to the file was released. This mimics the behavior under MS-DOS that once a user's application has exited, the machine can be turned off without data loss. This turned out to cause significant performance degradation, so instead we assigned this task to the update daemon. As the update daemon runs once every second, there can be at most a one-second delay after a user exits his or her applications until the machine is safe to power off. The original sync-on-close semantics are still available as a mount option for even greater reliability.

At first glance, checkpointing each writable file

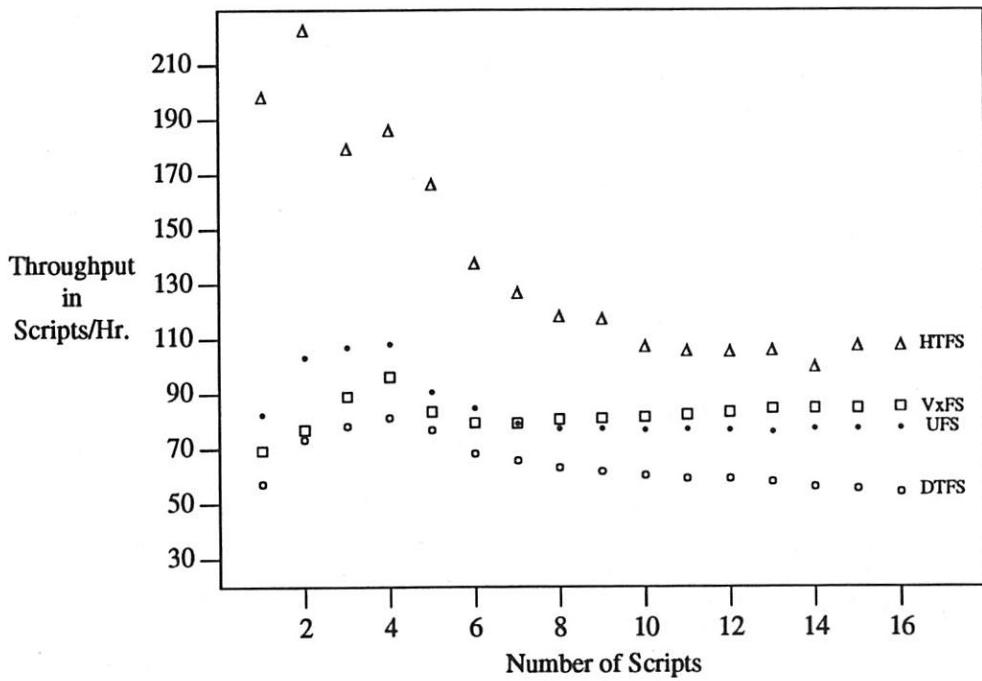


Figure 7. SDET Results

system every second appears to be an expensive proposition, but in practice it turns out to be relatively minor. Our studies have shown that the performance degradation in throughput benchmarks such as the SPEC consortium's SDET benchmark is on the order of ten percent when compared to running with the update daemon disabled. In addition, the update daemon only runs when there are recent file system modifications to be checkpointed, so if there is no DTFS activity, the update daemon overhead is zero.

6. Performance

We compared DTFS to other commercial file systems using the SPEC consortium's SDET benchmark. It simulates a software development environment, measuring overall system throughput in terms of the total amount of work that can be performed in a given unit of time. We think SDET is a good benchmark to measure file system performance because it models real-world system usage, which just happens to exercise the file system quite rigorously.

Our tests were run on an EISA-based 66MHz 80486 machine with an Adaptec 1742 SCSI disk controller in enhanced mode, a 5400 RPM disk with an average access time of 9.5 ms, and 32 MB of memory. We used UnixWare version 1.1 as the host operating system. Figure 7 summarizes the results of the benchmark using the UFS file system (with the file system parameters optimized to match the disk

geometry and speed), the HTFS file system (one of our commercial file systems, a UFS-compatible high-throughput file system using intent logging), the VxFS file system (the default UnixWare file system), and the DTFS file system.

DTFS did not perform as well as the other file systems, but its performance was still respectable. At 2 scripts, it performs almost as well as VxFS, and at 5 scripts it reaches 85% of the UFS throughput. At the peak, it attained 74% of the throughput of UFS, 83% of the throughput of VxFS, but only 43% of the throughput of HTFS. Originally we thought the performance loss was from the costs associated with compressing and decompressing user data. When we ran the benchmark with compression disabled, we were surprised to find that performance was slightly worse than with compression enabled. We attribute this to the additional I/O required when files are not compressed, and conclude that the CPU has enough bandwidth to compress and decompress files on-the-fly. The bottleneck in DTFS is the disk layout. We can overcome this by reorganizing our disk layout to perform fewer, but larger, I/O requests (a version number in the DTFS super block allows us to modify the layout of the file system and still support existing formats).

These conclusions are supported by the system timing results summarized in Table 1. The times are expressed in elapsed seconds to facilitate comparison.

Note that with all of the file systems, the benchmark spent about the same time at user level (ignoring rounding errors), which is what we would expect for identical workloads. DTFS spent the most time in the kernel, probably because of the compression and decompression. DTFS also had the most wait-I/O time.

File System	User Time	System Time	Wait-I/O Time	Idle Time
DTFS	24	61	88	0
HTFS	24	36	17	0
UFS	25	41	66	0
VxFS	24	51	67	0

Table 1. SDET Times in Seconds (4 Scripts)

We originally intended to illustrate the effect of the disparity between CPU and I/O speeds by repeating the benchmark with a slower disk drive, expecting the performance of DTFS to be closer to that of the other file systems. With the current disk layout it is not possible to meet this expectation, because the wait-I/O time for DTFS dominates the system time. Nonetheless, we are confident that once we reorganize the disk layout, we will be more able to support our belief that the overhead of performing data compression in the file system will become negligible as CPU speeds increase at a faster pace than I/O speeds.

Since the SDET benchmark measures overall system throughput, the effects of any one component, such as the file system, are diminished compared to benchmarks that isolate that component. For the purposes of comparison, we ran isolated file system throughput tests on the original hardware configuration. Our read test opens a file, optionally invalidates any pages from that file cached in-core, and reads the file. The read size and the total amount of data to be read are configurable. Similarly, our write test creates a file, and writes a data pattern to it in the increment specified, until the file reaches the requested size. Then the test optionally syncs the data to disk and invalidates the pages in-core.

We used a file size of 1 MB and two different I/O sizes to measure the raw throughput through each file system. The results are shown in Table 2. The first DTFS entry was derived using a pattern that only compresses by 25%. For comparison, the second DTFS entry shows the I/O throughput using a pattern that compresses by 85%.

For small writes, DTFS is between 2.0 and 4.0 times faster than VxFS, but is between 2.6 and 4.6 times slower than HTFS and UFS. For large writes,

File System	write 4096	read 4096	write 65536	read 65536
DTFS(25%)	235	366	269	373
DTFS(85%)	416	716	428	731
HTFS	1071	1870	1078	1687
UFS	1085	1860	1085	1678
VxFS	130	2490	471	1830

Table 2. Raw Throughput in KB/s

DTFS is between a factor of 1.1 and 1.8 times slower than VxFS, and is between 2.5 and 4.0 times slower than HTFS and UFS. For reads, DTFS is anywhere from 2.3 to 7.0 times slower than the other file systems, but this is hidden in a system-level benchmark where the page cache is active. Table 3 illustrates that when the page cache is used, DTFS is as fast as the other file systems (or, more correctly, the DTFS read vnode operation is as efficient as HTFS and UFS, and better than VxFS, since the reads are satisfied entirely by the page cache). This is one reason why system-level benchmarks like SDET show reasonable performance for DTFS (in other words, the page cache works).

File System	read 4096	read 65536
DTFS	6400	7800
HTFS	6400	7800
UFS	6400	7800
VxFS	6000	7300

Table 3. Cached Raw Throughput (KB/s)

Table 4 summarizes the effectiveness of the DTFS compression on different classes of files. The comparison is against a UFS file system with a block size of 4KB and a fragment size of 1KB. DTFS doesn't compress symbolic links, but the average size of a symbolic link is less than the average size of a symbolic link in a UFS file system. The 50% reduction in size is because a symbolic link requires one fragment (2 disk blocks) in UFS, and in DTFS it only requires 1 to 3 disk blocks, depending on the length of the pathname (short pathnames are stored directly in the inode, and most symbolic links on our system happen to be short).

When DTFS reports the number of disk blocks needed to represent a file, it adds one for the disk block containing the file's inode. Other file systems store multiple inodes per disk block, so cannot attribute the space to each file as easily. This tends to make comparisons of small files appear as if DTFS cannot compress them, and some even appear as if

File Type	Average Compression vs. 4K UFS
a.outs	36 %
C Source Files	37 %
Log Files	64 %
Man pages	44 %
Symbolic Links	50 %
X font binaries	53 %

Table 4. Sample Compression Statistics

they require more disk space with DTFS. This is misleading when you consider that DTFS doesn't have to waste disk space preallocating unused inodes.

For example, on a freshly-made 100000-block file system, UFS uses 6.0% of the disk for preallocated meta data. In comparison, VxFS uses about 7.2% of the disk, but DTFS only uses 0.052% of the disk for preallocated meta data.

The bar chart in Figure 8 illustrates the effectiveness of using DTFS as the default file system in a freshly-installed UnixWare system. For the other file systems, the default block sizes were used.

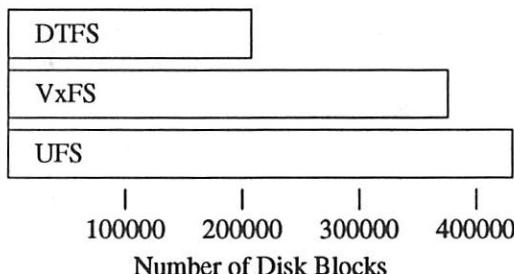


Figure 8. Sample Disk Usage

7. Administrative Interfaces

During the design of DTFS, we simplified several administrative interfaces, mainly because we believed that typical desktop and laptop users are not interested in becoming full-time system administrators, and even if they were, they probably didn't have the necessary skills. Thus, we modified several administrative commands to make them easier to use.

Our `fsck` doesn't prompt the user for directions on how to proceed. The DTFS `fsck` simply tries to fix anything it finds wrong. In addition, no `lost+found` directory is needed with a DTFS file system. Each DTFS inode contains a one-element cache of the last file name associated with the inode, along with the inode number for that file's parent directory. When `fsck` finds that an inode has no associated directory entry, `fsck` creates the entry

where it last existed. This frees users from having to use the inode number and file contents to figure out which files have been reconnected by `fsck`.

The DTFS `mkfs` command will automatically determine the size of a disk partition, freeing users from having to provide this information. An option exists to allow users to force a file system to be smaller than the disk partition containing it.

Our `mkfs` has no facility to allow users to configure the number of inodes for a particular file system. DTFS allocates inodes as they are needed, and any disk block can be allocated as an inode, so the only bound on the number of inodes for a given file system is the number of available disk blocks.

The DTFS `fsdb` command is much easier to use than the `fsdb` found with the S5 file system. (We don't expect most users to need `fsdb`, much less have the skills to use it, but some customers have requested that we provide it anyway.) The DTFS `fsdb` presents the user with a prompt, which by itself is a major improvement. The commands are less cryptic than their S5 counterparts. For example, the super block can be displayed in a human-readable form by typing `p sb`. To perform the equivalent operation with the S5 `fsdb`, one would have to type `512B.p0o`, and then interpret the octal numbers. The DTFS `fsdb` command also has facilities to log its input and output to a file and to print a command summary.

8. Lessons Learned

Although designed for use on small computer systems, it turned out that many customers (including our own development staff) use DTFS to store large source file archives. Although DTFS doesn't perform as well as other file systems, performance is good enough for most users so that this difference goes unnoticed.

8.1 Compatibility

One interesting problem resulted from our original inode allocation policy. For simplicity, a file's inode number is the block number of the disk block containing that file's inode. (An exception is made for the file system's root inode, which is hard-coded as 2 for historical reasons.) Since inodes can be allocated anywhere on disk, we were reaching the point where inode numbers greater than 65535 were being allocated. This can break older (SVR3) binary applications that use system interfaces that represent inode numbers as short integers, so we changed the inode allocation policy to bias itself in favor of lower-numbered disk blocks first. This didn't solve the problem entirely, but it made it less likely to occur.

Except for “.” and “..”, which are implicitly created when a directory is made, DTFS does not allow the creation of hard links to directories. We consider this to be an outdated feature since the advent of `mkdir(2)`, `rmdir(2)`, and symbolic links. In addition, it can wreak havoc with the file system hierarchy, and complicate file system implementations. We have found no compatibility problems with this policy.

8.2 System V Kernel Limitations

As stated in the overview, we found it necessary to provide our own routines to access the system’s buffer cache. The System V buffer cache interfaces that are used by file systems accept logical block numbers to identify disk blocks. The buffer cache routines then apply the formula

$$\text{physical block number} = \\ \text{logical block number} \times \text{block size}$$

to convert the logical block number into a physical block number, which is then associated with the disk buffer. This makes the routines useless to file systems that have a variable block size and already represent disk blocks by their physical block numbers. Thus, we found it necessary to provide routines that did not apply this formula.

Another deficiency we found with the System V buffer cache was the lack of a way to invalidate an individual buffer in the cache. A routine existed to invalidate an entire device, but no such routine existed that would invalidate a single disk buffer. DTFS needs to invalidate a disk buffer, if it exists, whenever a disk block is freed. This is because the disk blocks represented by that buffer might later be reallocated to a different sized extent. This could result in aliasing conflicts if the old disk buffer were to remain in the cache. Curiously enough, merely setting the `B_INVAL` or `B_STALE` flag in the buffer header does not prevent the `fsflush` kernel daemon from writing the buffer to disk, if the buffer had been previously delayed-written. Thus, besides an aliasing problem, valid data could be overwritten by stale data, so we had to write a routine to allow us to invalidate a single buffer.

9. Future Work

Our future plans for DTFS include adding new compression algorithms for different classes of data, improving performance, incorporating our HTFS technology to improve system throughput and speed up `fsck` time, and adding features that our customers need. An example of the latter is the ability to recover files removed or truncated accidentally (commonly known as “undelete”).

To date, DTFS has been ported to several versions of the Unix operating system, including SVR4, UnixWare, Solaris, and SCO Open Desktop. We intend to port it to other operating systems and hardware architectures.

10. Conclusions

With the disparity between CPU and I/O speeds, performing on-the-fly data compression and decompression transparently in the file system is a viable option. As the performance gap widens, DTFS performance will approach that of other file systems.

DTFS almost cuts the amount of disk space needed by conventional Unix file systems in half. In the introduction, we stated that DTFS was designed to be a commercial file system. The biggest sign of its commercial success is its planned inclusion into future versions of SCO’s Open Desktop product.

The architecture of DTFS lends itself well to the inclusion of intermediate data processing between the file system independent layer and the device driver layer of the Unix operating system. In fact, it only took an engineer a few days to convert DTFS from compressing and decompressing data to encrypting and decrypting data.

Acknowledgements

Many people have contributed to the ideas behind the design of DTFS, including George Bittner, Bob Butera, Ron Gomes, Gordon Harris, Mike Scheer, and Tim Williams. We would like to thank the many reviewers of this paper for their helpful comments, especially Margo Seltzer, who also provided several useful references.

References

- [BAC86] Bach, Maurice. J. *The Design of the UNIX Operating System*. Prentice Hall, Englewood Cliffs, NJ, 1986.
- [CAT91] Cate, Vincent and Thomas Gross. “Combining the Concepts of Compression and Caching for a Two-Level Filesystem,” *4th International Conference on Architectural Support for Programming Languages and Operating Systems*. Santa Clara, CA. April, 1991, pp. 200–211.
- [CHU92] Chutani, Sailesh, et al. “The Episode File System,” *Proceedings of the Winter 1992 USENIX Conference*. pp. 43–60.
- [COM79] Comer, Douglas. “The Ubiquitous B-Tree,” *Computing Surveys*. 11(2), June 1979, pp. 121–137.
- [EPP90] Eppinger, Jeffrey L. *Virtual Memory Management for Transaction Processing*

- Systems*. PhD Thesis #CMU-CS-89-115, Carnegie Mellon University, Pittsburgh, PA, 1989.
- [GRA81] Gray, Jim, et al. "The Recovery Manager of the System R Database Manager," *Computing Surveys*. 13(2), June 1981, pp. 223–242.
- [GRA93] Gray, Jim and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, 1993.
- [HAL94] Halfhill, Tom R. "How Safe is Data Compression?", *Byte*. 19(2), February 1994, pp. 56–74.
- [HIT94] Hitz, Dave, et al. "File System Design for an NFS File Server Appliance," *Proceedings of the Winter 1994 USENIX Conference*. pp. 235–246.
- [KAR94] Karedla, Ramakrishna, et al. "Caching Strategies to Improve Disk System Performance," *Computer*. 27(3), March 1994, pp. 38–46.
- [KLE86] Kleiman, Steven R. "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," *USENIX Association Summer Conference Proceedings, Atlanta 1986*. pp. 238–247.
- [MCK84] McKusick, Marshall K., et al. "A Fast File System for UNIX," *ACM Transactions on Computer Systems*. 2(3), August 1984, pp. 181–197.
- [OUS90] Ousterhout, John K. "Why Aren't Operating Systems Getting Faster as Fast as Hardware?", *Proceedings of the Summer 1990 USENIX Conference*. pp. 247–256.
- [SAN85] Sandberg, Russel, et al. "Design and Implementation of the Sun Network Filesystem," *USENIX Association Summer Conference Proceedings, Portland 1985*. pp. 119–130.
- [SEL92] Seltzer, Margo I. *File System Performance and Transaction Support*. PhD Thesis, University of California at Berkeley, 1992.
- [TAU91] Taunton, Mark. "Compressed Executables: an Exercise in Thinking Small," *Proceedings of the Summer 1991 USENIX Conference*. pp. 385–403.
- [WEL84] Welch, Terry A. "A Technique for High-Performance Data Compression," *Computer*. 17(6), June 1984, pp. 8–19.
- [ZIV77] Ziv, J. and A. Lempel. "A Universal Algorithm for Sequential Data Compression," *IEEE Transaction on Information Theory*. IT-23(3), May 1977, pp. 337–343.
- [ZIV78] Ziv, J. and A. Lempel. "Compression of Individual Sequences via Variable-Rate Coding," *IEEE Transaction on Information Theory*. IT-24(5), September 1978, pp. 530–536.

Trademarks

Adaptec is a registered trademark of Adaptec, Inc. MS-DOS is a registered trademark of Microsoft Corp. SCO and Open Desktop are registered trademarks of The Santa Cruz Operation, Inc. Solaris is a registered trademark of Sun Microsystems, Inc. Stacker is a trademark of Stac Electronics. UnixWare is trademark and UNIX is a registered trademark of UNIX System Laboratories, a wholly-owned subsidiary of Novell, Inc. VxFS is a registered trademark of VERITAS Corp.

Biographies

Morgan Clark is a Member of Technical Staff at Programmed Logic Corporation. Before joining PLC he worked at AT&T Bell Laboratories in Summit, NJ, and as a contractor at IBM. He has a B.A in Computer Science and Physics from Cornell University and an M.S. in Computer Science from the University of Wisconsin-Madison. His interests include operating systems, distributed systems, and performance analysis. He can be reached at morgan@prologic.com.

Stephen Rago is a Principal Member of Technical Staff at Programmed Logic Corporation. His interests include operating systems, networking, and performance analysis. He has a B.E. in Electrical Engineering and an M.S. in Computer Science, both from Stevens Institute of Technology. Stephen is the author of *UNIX System V Network Programming* (Reading, MA: Addison-Wesley, 1993). He can be reached at sar@prologic.com.

Sawmill: A High-Bandwidth Logging File System

Ken Shirriff

John Ousterhout

University of California, Berkeley

Abstract

This paper describes the implementation of Sawmill, a network file system using the RAID-II storage system. Sawmill takes advantage of the direct data path in RAID-II between the disks and the network, which bypasses the file server CPU. The key ideas in the implementation of Sawmill are combining logging (LFS) with RAID to obtain fast small writes, using new log layout techniques to improve bandwidth, and pipelining through the controller memory to reduce latency. The file system can currently read data at 21 MB/s and write data at 15 MB/s, close to the raw disk array bandwidth, while running on a relatively slow Sun-4. Performance measurements show that LFS improved performance of a stream of small writes by over a order of magnitude compared to writing directly to the RAID, and this improvement would be even larger with a faster CPU. Sawmill demonstrates that by using a storage system with a direct data path, a file system can provide data at bandwidths much higher than the file server itself could handle. However, processor speed is still an important factor, especially when handling many small requests in parallel.

1. Introduction

An I/O gap has arisen between the data demands of processors and the data rates individual disks can supply [PGK88]. This gap is worsening as processor speeds continue to increase and as new applications such as multimedia and scientific visualization demand ever higher data rates.

One common solution to the I/O bottleneck is the disk array, where several disks provide data in parallel. This can provide much higher performance than a single disk, while still providing relatively inexpensive storage. By using a RAID (Redundant Array of

Inexpensive Disks), disk arrays can be made reliable despite disk failures.

Even though disk arrays have the potential of supplying high-bandwidth data inexpensively, there are difficulties in making this data available to client machines across a network. For example, the RAID group at Berkeley built a prototype system called RAID-I, using a Sun-4/280 workstation connected to an array of 28 disks [CK91]. Unfortunately, the bandwidth available through the system was very low, only 2.3 MB/s, mainly because the memory system of the Sun 4 file server was a bottleneck.

To avoid the file server bottleneck, the Berkeley RAID group designed a storage system called RAID-II [DSH⁺94]. RAID-II uses hardware support to move data directly between the disks and the network at high rates, avoiding the bottleneck of moving data through the file server's CPU and memory.

Another problem with a RAID disk array is that small random writes are very expensive due to parity computation, which is used for reliability. One solution is a log-structured file system (LFS) [Ros92], which writes everything to a sequential log so there are only large sequential writes. Thus, a log-structured file system can greatly improve performance of small writes.

This paper describes the Sawmill file system, which has been designed to provide high bandwidths by taking advantage of the RAID-II architecture. Sawmill is a log-structured file system that is able to read data at 21 MB/s and write at 15 MB/s, close to the raw disk bandwidth. Sawmill is designed to operate as a file server on a high-bandwidth network, providing file service to network clients.

Sawmill solves two problems: how to make use of a direct data path and how to combine a log-struc-

tured file system with a disk array. The key ideas in Sawmill are using streaming instead of caching, performing log layout on the fly, minimizing per-block overheads, and moving metadata over a separate control path.

The remainder of the paper is organized as follows. Section 2 gives some background on RAID storage, the RAID-II storage system, and log-structured file systems. Section 3 describes in more detail the implementation of the Sawmill file system and the techniques it uses to obtain high bandwidth. Section 4 contains performance measurements of Sawmill. Section 5 discusses related work, Section 6 discusses future work, and Section 7 concludes the paper.

2. Background

There are three main concepts combined in the Sawmill file system: the RAID disk array, hardware support for a fast data path, and the log-structured file system (LFS). This section gives background information on these ideas.

2.1 RAID

A RAID (Redundant Array of Inexpensive Disks) combines two ideas: parallelism and redundancy [PGK88]. A RAID uses multiple disks in parallel to provide much higher bandwidth than a single disk. A RAID can also perform multiple operations in parallel. Redundancy in the form of parity is used to maintain reliability. By storing parity, data can be fully recovered after a single disk failure.

In a RAID, data is striped across multiple disks. We use a RAID-5 architecture, which stripes data as blocks. With N disks, a group of $N-1$ data blocks is striped across $N-1$ disks. A parity block is computed by exclusive-oring these $N-1$ blocks and the parity block is stored on the remaining disk. Parity is distributed; successive parity blocks are stored on different disks to avoid the bottleneck of a dedicated parity disk. Each individual block is called a *stripe unit*, and a collection of $N-1$ data blocks and a parity block is called a *parity stripe*.

For peak efficiency, a full parity stripe should be written at once. In this case, the data and the parity can be written out in parallel. If only part of the parity stripe is modified, parity must be recomputed from the data already on disk. This overhead is especially costly for small writes. Figure 1 illustrates how parity is recomputed after a small write: the old data and the recomputed parity are written, resulting in 4 disk

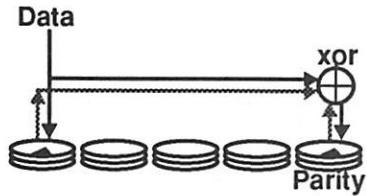


Figure 1. Read, modify, write. When a small part of the stripe is written to disk, the new parity is computed from the old data, the old parity, and the new data. Thus, four disk operations are required.

operations. Thus, small writes are relatively expensive with a RAID, and writes of a parity stripe are most efficient.

2.2 The RAID-II storage architecture

The Berkeley RAID project found that a high-bandwidth disk array could easily saturate the memory bandwidth of a typical workstation file server [CK91]. The RAID-I prototype used a Sun-4/280 workstation connected to an array of 28 disks. The bandwidth available through the system was very low, only 2.3 MB/s, mainly due to the low bandwidth of the Sun 4 file server's memory system and backplane.

To avoid the file server bottleneck, the follow-on RAID-II project built a storage system with a new hardware architecture designed to support disk array bandwidths. This hardware provides a high-bandwidth data path between the disks and the network, bypassing the file server. This results in separation of the control and data paths: the file server handles requests and provides low-bandwidth control commands, while the controller board provides high-bandwidth data movement. The controller also has fast memory that can be used for buffering and prefetching.

Figure 2 illustrates the hardware configuration of RAID-II. The controller provides a fast path between the disks and the network. It is built around a high-bandwidth crossbar switch that connects memory buffers to various functional units. These units include a fast HIPPI network interface, an XOR engine for computing parity, and the disk interfaces. Although RAID-II was designed to support 40 MB/s, the maximum bandwidth is currently about 24 MB/s. Performance is currently limited by the disk controller and the VME link to the controller; each of the four disk controllers can handle about 6 MB/s. We have used a HIPPI network and an Ultranet to connect to clients. Further details of RAID-II are given in [DSH⁹⁴].

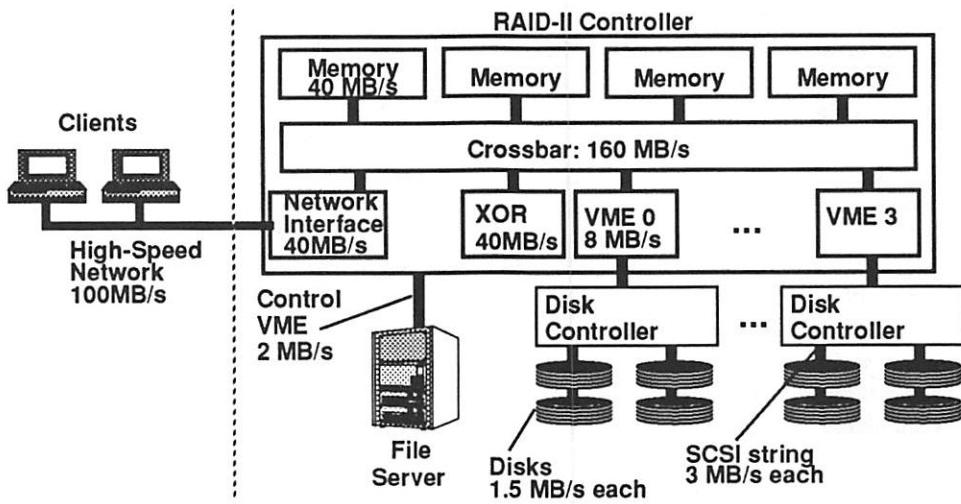


Figure 2. The RAID-II storage architecture. The RAID-II storage system (on the right) provides high-bandwidth file access to the network clients (on the left). The RAID-II controller provides a fast data path between the disks and the network. The controller is built around a fast crossbar that connects the disks, memory, and the network. The file server, running the Sawmill file system, controls the storage system but stays out of the data path. The current configuration uses 16 disks on four controllers. These disk are treated as a single RAID stripe group.

2.3 Log-structured file systems

The third idea used by the Sawmill file system is the log-structured file system (LFS). A log-structured file system [Ros92] writes data only to a sequential log. (A file block does not have a fixed position on disk, but instead its position changes every time the block is rewritten.) The log is written to disk in large units, called *segments*, on the order of 1 MB in length. As the log fills the disk, old segments are read in to free up the stale data; this process is called cleaning.

Because it writes only to the log, an LFS does not perform any random writes, but only large sequential log updates. This is an advantage with any disk system because it avoids the seek time and rotational latency of a random write. However, there is an even more significant advantage with a RAID due to the high parity overhead of small writes to RAID. By using an LFS, random writes can use the full sequential write bandwidth of the disk array because they are grouped together into a log write. The LFS segment size must be a multiple of the parity stripe size for maximum performance.

3. Implementation

This section describes the implementation of the Sawmill file system. Section 3.1 gives an overview of how Sawmill processes client requests. Section 3.2 describes how Sawmill uses the controller memory. Section 3.3 explains how read performance is improved. Section 3.4 describes the new log layout

techniques. Section 3.5 describes the handling of metadata.

3.1 Handling a typical request

Clients communicate with Sawmill over a high-speed network. In the current implementation, client applications are linked with a small library to use the Sawmill file system. This library converts file system operations into socket operations. For instance, to open a file, the client calls a library routine, which opens a socket connection to the Sawmill file server. The client then sends the open parameters through the socket and receives the status from the file server. It would also be straightforward to provide access to data via NFS; however, performance would be limited by NFS protocol overhead.

For a read, the client library routine sends the read parameters over the socket and then receives the data. On the server side, the read request is received into controller memory and copied through the RAID-II controller into the file server. The Sawmill file system determines where the data is stored on disk. The file server sends commands to the RAID-II controller to read data from disk into controller memory. When the first block of data has arrived from the disks, the file server issues commands to send the data across the network and to read the next blocks of data. Pipelining the transfers in this way reduces latency. Note that the file server only processes requests and handles control messages, while the RAID-II controller does the actual data movement.

Writes to Sawmill are handled in a similar fashion. The client sends the write parameters over the socket, followed by the data. The file server receives the write request, determines where in controller memory to receive the data, and informs the RAID-II controller to start receiving data. After a full segment of the log has been collected in controller memory, the file server commands the controller to write the data to disk. Again, the data is transferred from network to disk without passing through the file server's memory.

Sawmill currently uses the same network to receive requests and to transfer data. However, it would be straightforward to use separate control and data networks. This would typically be used if the high-speed network had high latency, for instance, to set up a connection.

3.2 Controller memory

The RAID-II controller board has a 32 MB memory buffer. This memory is used by the RAID striping driver and the Sawmill file system. The RAID striping driver uses memory for buffering, to compute parity, and to reconstruct data after a disk failure. The Sawmill file system uses the remaining memory for several purposes: buffering network requests and replies, holding LFS segments before they are written to disk, buffering data during reads, and holding LFS segments for cleaning. This section describes the use of controller memory for reads. Section 3.4 will describe the use of controller memory for layout of write data in the log.

In the original Sawmill file system design, we planned to implement a standard file system block cache in controller memory. However, we decided against this for two reasons. First, a block cache requires a significant amount of CPU overhead for each block, to check whether a block is in the cache and to update the cache information with new blocks. Second, with RAID-II, the memory available for a cache would be about 20 MB; we expect this is too small to provide a high hit rate with high-bandwidth applications. Note that with a data rate of about 20 MB/s, the lifetime of cached data would be only one second. In any case, client caching could provide most of the potential benefits of a server data cache.

Currently, the Sawmill file system uses controller memory for pipelining read requests. That is, for large reads, disk operations are overlapped with network operations to hide the latency of disk and network operations. As data blocks are read from disk to memory, previous blocks are being sent from memory to

the network. We are currently adding prefetching to Sawmill to obtain this benefit for small operations.

3.3 Batching of reads

For efficiency, data should be read with a single disk operation whenever possible. Unfortunately, a sequential request might not be stored sequentially in the log. Fortunately, there will often be locality of writes, so the data may be stored in the same segment, even if the blocks are not stored sequentially. In this case, it would be more efficient to read the entire segment and then send the pieces from memory over the network in the proper order, rather than perform multiple disk operations to fetch the blocks in order.

To group reads, the file system loops over each block, checking its position on disk and seeing whether it is part of the same segment. When it collects as many blocks as possible, it can read them all at once. More complex algorithms are possible to improve performance when reading out-of-order data. For instance, batching may be combined with prefetching to read blocks before they are requested.

3.4 On-the-fly layout

A log-structured file system must arrange new file data into the sequential log that is written to disk; this process is called log layout. Sawmill uses a new method of performing log layout, called on-the-fly layout, to get more efficiency. In a standard LFS implementation [Ros92][SBMS93], write data is stored in the file system block cache. A backend process pulls blocks out of the cache and places the blocks in the log. In contrast, Sawmill assigns a position in the log to each data block when the write request comes in. The file system informs the controller of this position, causing incoming data to go directly from the network into the proper position in the log. At the same time, the file system reserves space in the log for any needed metadata such as inodes. Figure 3 illustrates the two techniques.

There are several advantages of doing layout when writes are received rather than as a backend process. First, layout for a large write can be done as a single operation, instead of many per-block operations, thereby minimizing the processing overhead. Second, blocks do not have to be moved to and from the cache, but are immediately put in the proper location. The RAID striping driver requires the segment written to disk to be stored contiguously in memory. Thus, a copy operation would be required, which would significantly reduce performance. (An alternative would be a device driver that accepts a scatter-

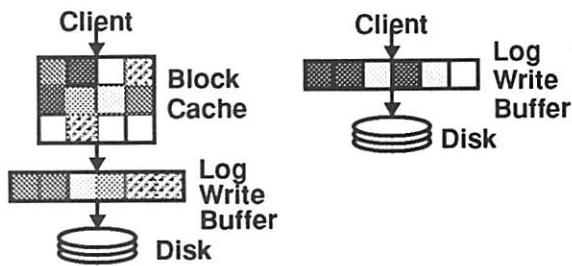


Figure 3. Comparison of log layout techniques. On the left, the layout uses a cache as in traditional log-structured file systems. On the right, the layout is done on the fly, directly into a write buffer.

gather array so layout could be done by moving pointers instead of blocks.) On-the-fly layout had a large impact on performance of Sawmill; an earlier implementation used a block cache for writes but due to copying and per-block overheads, performance was limited to under two megabytes per second.

Because it doesn't use a cache, on-the-fly layout has a few potential disadvantages. First, data cannot be reorganized before it is written. For instance, if writes to a file are received in random order into a cache, they could be taken out of the cache sequentially and written in order. However, if blocks are placed in the log immediately, they will be written in the random order in which they were received. This will decrease performance if they are later read sequentially. The second disadvantage occurs if data blocks are modified or deleted. With a cache, only the live blocks will be written to disk. However, with on-the-fly layout, the dead blocks will have a position in the log and will go to disk, resulting in unnecessary disk traffic. We expect that these disadvantages will not be important in practice. Because of the high data rates, blocks are likely to go to disk before they would be modified or deleted. Also, if there were a cache on the clients, the client cache could do the processing and provide the benefits.

3.5 Metadata movement

Besides handling file data, the file system must handle various types of metadata such as information on files (inodes), information about where blocks are on disk, the directory structure of the file system, and information about the contents of LFS segments. With a normal storage system architecture, the file system just accesses metadata as it needs it. However, with RAID-II any required metadata must be explicitly copied between file server memory and the RAID-II controller. Since moving data over this path is rela-

tively slow, metadata is cached in file server memory. As a result, the Sawmill file system must maintain consistency among metadata in file server memory, metadata in controller memory, and metadata stored on disk.

4. Performance measurements

This section describes performance measurements of the Sawmill file system. Section 4.1 describes measurements of a single request stream with requests that are handled sequentially. Section 4.2 describes measurements of multiple requests processed in parallel: these numbers indicate the additional performance that a disk array can obtain by handling concurrent requests. Performance results are given for the raw disk array, the disk array configured as a RAID, and the disk array running Sawmill.

These performance measurements used RAID-II with 16 disks. The stripe unit was 64 KB, yielding a parity stripe size of 960 KB. In other words, writes of 960 KB transfer 64 KB data blocks to 15 disks and a 64 KB parity block to 1 disk in parallel. The LFS segment size was also 960 KB. The measurements in this section are of random requests of various fixed sizes. For these measurements, the Sawmill file server was a Sun-4/280, which is relatively slow (about 9 SPEC89 integer SPECmarks).

Because we don't have a client that can handle the data rates of RAID-II yet, the following performance tests did not transmit data across the network, but only transferred the data between the disks and controller memory. Our current client can only transfer 3 to 4 MB/s over the network. However, the measured server CPU load to handle this network traffic was about 2%. Thus, the server could transfer the full RAID-II bandwidth over the network without a CPU bottleneck arising due to the network traffic. Therefore, the performance numbers in this section should be a reasonable indication of the actual network performance with a fast client.

There are several key results from the performance measurements. First, Sawmill is able to provide about 80% of the raw disk bandwidth for large requests. Sawmill is also able to handle a single stream of small read requests with performance close to that of the raw disk. Because of LFS, Sawmill can handle small writes an order of magnitude faster than the raw disk. However, the server CPU is a performance bottleneck for small writes and for multiple request streams. Small write performance with Sawmill would improve dramatically with a faster CPU.

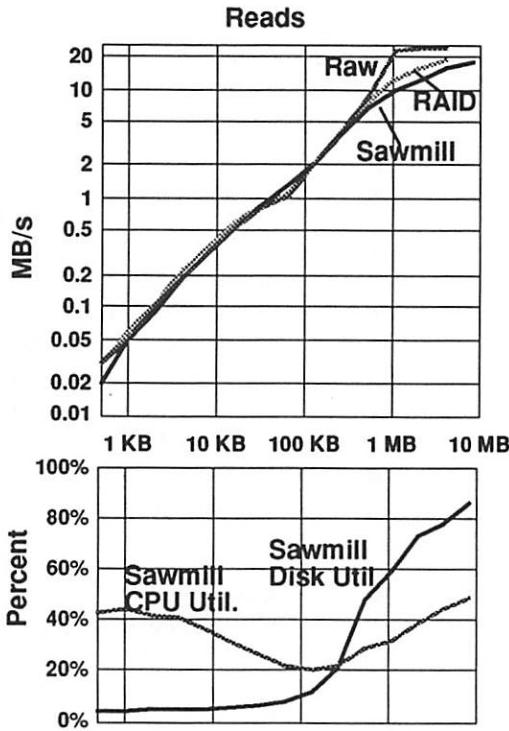


Figure 4. Read performance for a single stream of random requests as a function of the request size. The top graph shows bandwidth through the raw disk array, the disk array treated as a RAID, and the Sawmill file system. The lower graph shows CPU utilization and disk utilization through Sawmill.

Also, Sawmill can only support about 3 or 4 concurrent streams of small requests before the CPU becomes a bottleneck.

4.1 Single request stream

This section describes the performance of the raw disk array, the disk array treated as a RAID, and the Sawmill file system, when receiving a single stream of requests. This indicates the performance available to a single application. Figure 4 shows read requests and Figure 5 shows write requests.

4.1.1 Raw disk array performance

To understand the performance, the first item to examine is the disk array. The array contains IBM 0661 disks [IBM91]. These disks have average rotational latency of 7 ms and average seek time of 12 ms. Each disk can read and write at about 1.6 MB/s. The disk array was configured with 16 disks on four controllers. A single SCSI string was able to read at 3.14 MB/s with two disks. With two strings of two disks per controller, each controller can provide 6.24 MB/s. For the measurements in this section, the disk array

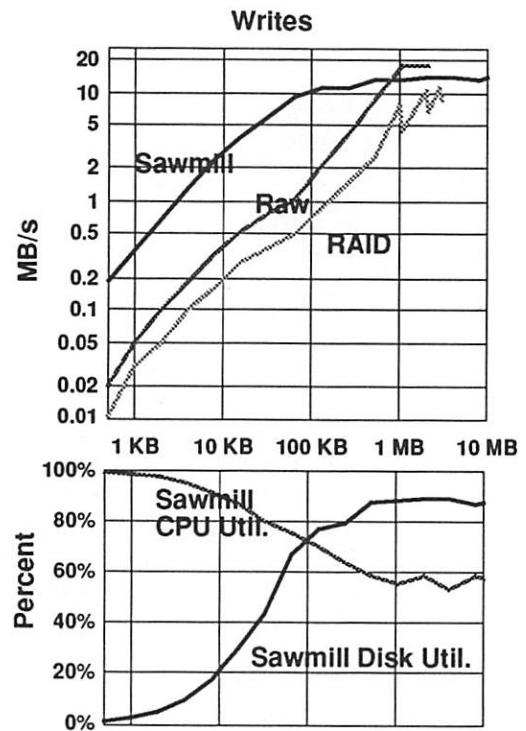


Figure 5. Write performance for a single request stream. As in Figure 4, the top graph shows performance of the raw disk array, the RAID, and Sawmill. The lower graph shows CPU and disk utilization with Sawmill. Note that because of LFS, Sawmill performance is much better than writing directly to the disks. Also note that small requests are totally CPU limited. This is due to LFS: for small writes, almost all the time is spent grouping them together. CPU utilization remains high, even for large writes.

was treated as independent disks, with no parity computation. To make the measurements comparable to the RAID and Sawmill measurements, requests were broken into 64 KB blocks and made to the appropriate number of disks: requests smaller than 64 KB used a single disk and larger requests used multiple disks.

The “Raw” lines in Figures 4 and 5 show the performance of the raw disk array as a function of request size. Performance was highly dependent on request size. Peak read performance was 24.5 MB/s and peak write performance was 18.4 MB/s. For small requests, seek time dominated.

With the raw disk array, the processing overhead is minimal, so the performance is almost entirely dependent on the disk speed. The time to complete a request to a disk can be modeled approximately as:

$$20 \text{ ms} + \text{disk_request_size} / \text{peak_bandwidth}$$

where the measured peak bandwidth is 1.57 MB/s for reads and 1.16 MB/s for writes. By dividing the request size by this time, the bandwidth for a given request size can be modeled. Writes are slower

because of missed revolutions; the disks have a track buffer that minimizes missed revolution costs for reads.

4.1.2 RAID performance

In the previous section, the disk array was viewed as independent disks. In this section, the disks are treated as a RAID, but without a file system. The RAID striping driver receives requests and breaks them up into stripe units spread across multiple disks. For writes, parity is computed. These measurements are significant because Sawmill is built on top of the RAID.

The “RAID” lines in Figures 4 and 5 show the read and write performance of a single stream to the RAID device. Read performance is generally the same as for the raw disk. For large accesses there is a small performance loss, mainly from processing overhead and increased latency due to the RAID striping driver. RAID writes take at least twice as long as writes to the raw disk due to the read-modify-write parity update. Note the spikes in write performance at multiples of 960 KB; these sizes are multiples of the parity stripe size and thus don’t require a parity read-modify-write.

These measurements show that for good write performance, the file system must perform operations in multiples of the parity stripe size.

4.1.3 Sawmill performance

The Sawmill file system, by using the techniques described earlier, was able to obtain peak bandwidth of about 80% of the raw disk bandwidth. For the measurements in this section, random reads and writes of various sizes were made to a single 30 MB file.

For a single request stream, Sawmill read performance is close to the raw disk performance. There is a performance loss for very large requests due to the per-request file system overhead of Sawmill and due to reads broken across multiple LFS segments. For large sequential reads, maximum performance is 21 MB/s, slightly below the raw disk bandwidth. For individual small random reads, as shown in Figure 4, performance is limited by the file system overhead and the seek time of the disks, resulting in a minimum latency of about 20 ms per operation.

Figure 4 shows the CPU and disk usage for Sawmill read requests. The disk usage shows the percentage of time the disks are in use, averaged across all 16 disks (e.g. 8 disks in use 50% of the time is 25% utili-

zation). Because the measurements in Figure 4 use only a single stream of requests, small requests only use one disk at a time and cannot exceed 6.25% utilization. Note that CPU load is fairly high in Figure 4, but drops around 100 KB. This shape is due to two factors: CPU overhead per byte and the total number of bytes. The CPU usage per disk operation is roughly constant. As the request size increases to the 64 KB stripe unit, each disk operation transfers more data, so the fraction of time spent with CPU overhead decreases. However, as request size continues to increase beyond 64 KB, more disks are used in parallel, causing the CPU usage to climb again.

Figure 5 shows write performance. Large files can be written to disk at about 15 MB/s. Note that small writes are totally CPU limited. This is due to LFS: since small writes are batched together and written sequentially, very many small writes take place before a disk operation occurs. Thus, the file system overhead to perform this batching dominates small write performance. As the request size increases, per-operation CPU time becomes less important, causing disk usage to climb and CPU usage to drop. These performance measurements omit the cost of writes that modify less than a file block; in this case an additional read would be required to fetch the unmodified data.

Figure 5 illustrates the benefits of a log-structured file system and also shows the performance lost due to CPU load. Because LFS groups small writes together, performance is about 20 times that of the RAID. This performance would be much better with a faster CPU; if there were no CPU load, the Sawmill line would be approximately flat around 15 MB/s, since the segment size written to disk is fixed. A more realistic projection is to consider a CPU that is ten times faster, such as a DEC Alpha. Small write performance would scale almost linearly, since small writes are totally CPU bound. This would result in small write performance of 2 MB/s, about 200 times the performance of writing directly to the RAID.

Figure 5 can be used to estimate the write performance of a file system based on the Unix FFS [MJLF84] running on RAID-II and compare it with Sawmill. If writes go to disk individually, the Unix file system would have performance similar to that of the RAID (assuming no CPU overhead for the Unix file system and ignoring the seeks to write metadata in the Unix file system), so Sawmill would be a factor of 20 faster for small writes. A Unix file system with clustering [MK91] would shift performance along the RAID line, since the writes would be in larger units.

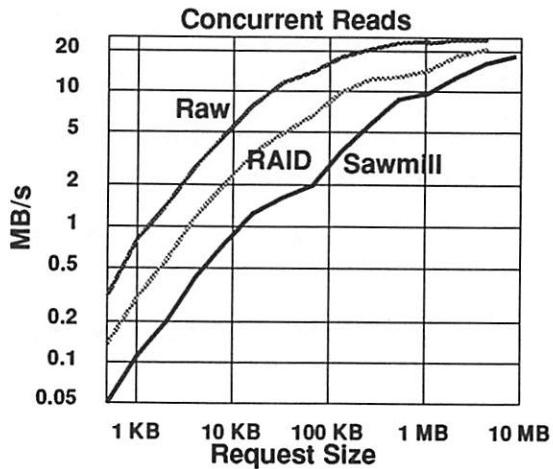


Figure 6. Read performance with multiple concurrent requests. The lines indicate the raw disk array as 16 independent disks, the disk array treated as a RAID, and the Sawmill file system. Concurrency of RAID and Sawmill are limited by the CPU load.

However, Sawmill performance would still be much better because Sawmill writes are grouped into full parity stripes. With a faster processor, the benefits of Sawmill would be even more dramatic because small writes are currently CPU limited. Sawmill would have additional overhead due to cleaning (which will be discussed in Section 6.1), but the performance benefit of logging is likely to greatly exceed this cost.

4.2 Multiple stream performance

This section describes the performance of the system when multiple requests are processed in parallel. This indicates the performance if, for example, multiple applications or multiple clients were using the system. Figure 6 shows read performance with concurrency and Figure 7 shows write performance.

A disk array can process multiple requests in parallel if they go to separate disks. This improves the overall system throughput and the total number of operations per second, but doesn't improve the performance of any particular request stream. Concurrency is a significant benefit for small requests, which only use a single disk. Potentially, 16 small reads or 8 small RAID writes could take place in parallel, improving performance by the same factor. Requests larger than the 64K stripe unit will be striped across multiple disks and become inherently parallel. Thus, multiple requests improve the performance much more for small requests than large requests.

The limiting factor to concurrency in our system is the CPU load. Unfortunately, our file server CPU is relatively slow and cannot handle the full potential

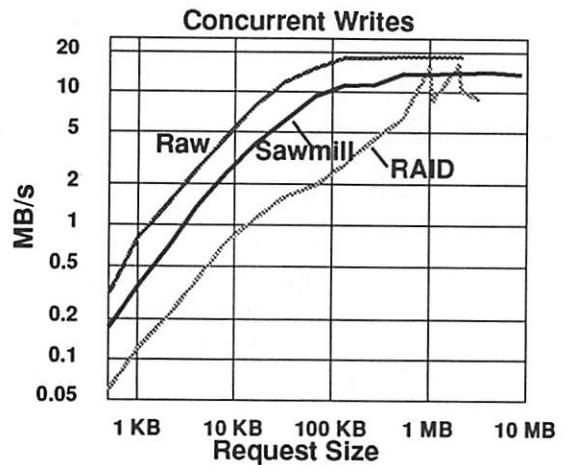


Figure 7. Write performance of the raw disk array, RAID, and Sawmill with multiple concurrent requests. The spikes in RAID write performance at 960 KB and 1.92 MB illustrate the write performance possible when a full stripe is written at once; other writes require an expensive parity update. Note that because of LFS, the performance of Sawmill exceeds performance of the RAID.

concurrency of the system. Only 3 or 4 Sawmill operations can be run in parallel before the CPU becomes a bottleneck. Since the disk array could support 16 operations in parallel, small operation throughput would increase substantially with a better processor.

In Figures 6 and 7, the degree of concurrency used depended on how much concurrency the system could handle before saturating. The raw disks handled 16 independent request streams. The RAID handled 8 independent streams for small requests, with the number decreasing as the request size increased. Sawmill handled 4 concurrent small read requests. For Sawmill, no write concurrency was required because LFS results in all the disks being used in parallel.

For the raw disk array, the CPU load is almost entirely determined by the number of disk operations. Starting a disk operation, handling the interrupt, and concluding the disk operation take about 1 ms in total. Since the SCSI controller can only handle requests up to 128KB, a larger logical operation results in multiple disk operations. Due to these factors, CPU load for 16 parallel raw disk operations is about 70% for small operations, drops to 20% for 128 KB requests, and then climbs slightly. This indicates that although the RAID-II architecture allows data rates greatly exceeding what the file server could move, our current file server CPU is just barely able to handle the high interrupt rates of small disk operations.

Besides bandwidth, another important measurement is the total number of I/O operations the disk array can support per second. Figure 8 shows the

Type	Read ops per sec	Write ops per sec
1 disk	50	45
16 disk array	710	670
RAID: 1 request	55	25
RAID: 8 requests	300	100
Sawmill: 1 req.	45	345
Sawmill: 4 reqs.	95	345

Figure 8. Random 4KB I/O operations. This table shows the number of I/O operations RAID-II can handle per second under various configurations. CPU load limits multiple RAID and Sawmill requests.

number of I/O operations per second the system can support, and compares a single request stream with multiple request streams. These measurements are for 4 KB random operations.

In theory, by performing multiple operations in parallel, performance should scale with the number of disks. However, the main limitation on parallelism with RAID-II is the CPU load; handling multiple requests through RAID or Sawmill saturates the CPU before all disks are in use. The RAID striping driver has more processing load than the raw disk, and the Sawmill file system has additional load. Thus, the number of parallel requests possible before the CPU saturates is reduced. The RAID measurements were saturated with 8 requests in parallel, while the Sawmill measurements were saturated with 4 requests.

For a single raw disk, the number of operations per second is almost entirely limited by the 7 ms rotational latency and the 12 ms average seek time. With 16 raw disks, Figure 8 shows about a 5% performance penalty; the CPU load is about 75%, so there is some delay in starting requests.

When the disk array is treated as a RAID, the I/O rates change significantly. A single stream of read requests to the RAID uses a single disk, so performance is approximately the same as for a single disk. (Performance in Figure 8 is slightly better for the RAID because seek distance was shorter for the striped requests.) Performance of a single small write stream is about half of the read performance due to the read-modify-write parity update. With multiple parallel request streams, the RAID uses disks in parallel. Performance levels off around 8 request streams because the CPU becomes saturated. A faster CPU would significantly increase the total number of I/Os per second that the RAID can support.

Figure 8 also shows performance of the Sawmill file system for small requests. Comparing the read measurements to the RAID with a single request

stream shows that there is a slight performance loss due to Sawmill; this is due to overhead in the file system. (The read latency is simply the reciprocal of the I/Os per second, or about 22 ms.) Due to the CPU load, reads obtained little benefit from concurrency. For writes, the number of I/Os per second is very good compared to the RAID, showing the benefit of LFS. With a faster CPU, the number of I/Os per second would be even higher, since performance is CPU limited, not disk limited.

4.3 Scalability

This section describes how the performance measurements are likely to change with improvements in technology. To summarize, performance of large requests, small writes, and concurrent operations is likely to improve with faster technology. However, small reads are limited by the performance of individual disk drives, which will not improve as quickly over time.

Because small writes are limited by CPU speed, small write performance will scale almost linearly with increases in processing power. Current high-performance workstations already have ten times the CPU power of our Sun-4 server, and even faster machines will become available over the next few years. Thus, the performance of small writes in a Sawmill-like file system should improve dramatically over what we measured.

Small individual random reads are limited by disk positioning time. Disk positioning times are likely to improve relatively slowly, compared to improvements in CPU speed. Faster CPU speeds will improve the potential parallelism of small reads, which was limited by our CPU. Increasing the number of disks will increase the total number of independent reads that can be carried out simultaneously. This will increase the number of I/O operations per second, but won't help the performance of an individual request. Prefetching can be used to improve the performance of individual requests; if the data is fetched before it is required, the disk latency will not affect latency to handle the request. One technique for this is Gibson's Transparent Informed Prefetching [GPS93]; by obtaining hints from the application, the file system can fetch data before it is required.

Large reads and writes already achieve close to the raw system performance with Sawmill. Thus, large request performance will only increase with storage systems that have more or faster disks. The CPU speed will have to increase proportionally or else the CPU will become a bottleneck. However,

note that the current server is a Sun-4 and much faster CPUs are readily available.

5. Related work

There are several storage systems that provide high-performance I/O. One fundamental problem these systems must solve is how to provide the data without being limited by the file server's CPU, memory, and I/O bandwidth. There are various methods that have been used to solve these problems. For instance, mass storage systems usually use an expensive mainframe, designed to support high bandwidth, as the file server. Other systems use multiprocessors or multiple servers to provide sufficient server power.

5.1 Mass storage systems

There are several high-performance mass storage systems in use at supercomputing centers. One example is the LINCS storage system at Lawrence Livermore National Laboratory [HCF⁺90]; this storage system has 200 GB of disk connected to an Amdahl 5868. Another example is a system at the Los Alamos National Laboratory [CMS90], which has an IBM 3090 running the Los Alamos Common File System.

These systems solve the problem of the file server being a performance bottleneck by using a mainframe as the file server. The mainframe server is designed to have the I/O bandwidth, memory bandwidth, and CPU power necessary to provide very high data rates to clients. In addition, the server is connected to a network sufficient to handle very high data rates. The disadvantage of these mass storage systems is their cost. Because they are custom-designed and use an expensive mainframe, they are used at very few sites.

5.2 Multiprocessor file servers

Another approach to increasing I/O performance is to use a multiprocessor as the file server. Such a system avoids the problem of the server being a performance bottleneck by using multiple processors, with the associated gain in server CPU power and memory bandwidth. One example of this is the Auspex NFS server [Nel90]. The Auspex system uses asymmetric functional multiprocessing, in which separate processors deal with the Ethernet, files, disk, and management. The necessary disk bandwidth is provided by parallel SCSI disks. However, the performance of the Auspex is limited by its use of NFS, Ethernet, and a single 55 MB/s VME bus; measurements show it can supply about 400 KB/s to a single client and can sat-

rate an Ethernet network with 1MB/s per Ethernet connection [Wil90].

The DataMesh project proposed a different approach to multiprocessing [Wil91]. The proposed system would consist of a large array of disk nodes, where each node had a fast CPU (20 MIPS) and 8 to 32 MB of memory. These nodes would be connected to a high-performance interconnection network.

By providing multiple processors, these systems avoid bottlenecks from limited server CPU bandwidths. However, this tends to be an expensive solution, since it requires buying enough processors to provide the necessary memory bandwidth. Even so, the Auspex server still has a memory bandwidth bottleneck since it uses a single VME bus.

5.3 Striping across servers

High bandwidth access to very large data objects can also be provided by striping files across multiple servers, so that overall bandwidth isn't limited by the memory system of a single server. An example is the Swift system [CL91]. In this system, data is striped across multiple file servers and networks to provide more bandwidth than a single server could provide.

A second system that stripes data across multiple servers is Zebra [HO93]. In Zebra, each client writes its data to a sequential log. This log is then striped across multiple servers, each with a disk. Zebra and Sawmill both combine LFS and RAID. The key difference is that Zebra uses multiple servers with single disks and Sawmill uses a single server with multiple disks. Swift and Zebra avoid the file server bottleneck by using multiple servers, while Sawmill avoids the bottleneck by using a data path in hardware. There is a cost trade-off: Sawmill requires a special controller, while Swift and Zebra require multiple fast servers.

5.4 RAID parity updates

There are several techniques to reduce the cost of updating parity after a partial stripe write. One technique is Parity Logging [SGH93]. In this technique, parity updates are written to a log. At regular intervals, the log is scanned and the parity modifications are applied to the standard RAID parity blocks. Floating Parity [MRK93] is a second technique for minimizing parity cost. In this technique, multiple parity blocks are reserved on disk. Updates can use the block closest to the current rotational position of the disk. The Logical Disk [dJKH93] implements a log-structured file system at the disk level rather than the file system level by writing all blocks sequentially to a log.

and maintaining the mapping between logical disk blocks and physical disk blocks. Since it writes blocks to a sequential log, it avoids random parity updates.

6. Future Work

Probably the most important unanswered issue with Sawmill is the cost of cleaning a log-structured file system. Another major issue is how to improve the performance of small reads; this was discussed earlier. A related issue is improving large read performance by data reorganization.

6.1 Cleaning cost

One of the key unanswered questions about Sawmill is the overhead due to log cleaning. Cleaning is the process of garbage-collecting the log to free up space. Cleaning is not yet operational in Sawmill, so performance measurements are not available. Previous work [Ros92] indicated that overall cleaning costs would be low. However, [SBMS93] found cleaning costs to be high for some workloads, such as transaction processing. Costs were high particularly in environments with largely full disks, and cleaning could potentially cause service interruption. The former work did not have dynamic performance measurements of the cleaner, and the latter measurements were on a largely untuned implementation.

The impact of cleaning on the performance of log-structured file systems continues to be a topic requiring additional investigation, which is beyond the scope of this paper. At this point, there is no evidence to suggest that Sawmill will not perform well for the workloads typical of office environments. Furthermore, even if cleaning costs are high, Sawmill's ability to avoid the "small-write problem" of RAID devices could well offset such overheads.

6.2 Data reorganization

Log-structured file systems are write-optimized; that is, they organize data for fast writes, even though this may disadvantage later reads. In particular, after random writes data may not be stored sequentially on disk, resulting in lower performance for later sequential reads. However, by reorganizing data on disk so the data is sequential, the reorganized data can be more efficient for reads. Thus, reorganization on Sawmill could improve sequential read performance. This reorganization could either take place during idle times in the system, or it could be integrated with cleaning so cleaned data is written back in a better pattern for reads.

7. Conclusions

This paper has described Sawmill, a high bandwidth file system that uses the RAID-II disk array. Sawmill uses a cost-effective file server to provide high-bandwidth access to a disk array. By taking advantage of hardware support, Sawmill provides data rates much higher than the memory bandwidth of the file server.

Measurements show that for large requests Sawmill operates at close to the raw bandwidth of the disk array, reading data at a peak rate of 21 MB/s and writing data at 15 MB/s, while running on a Sun-4. The log-structured file system improved performance of a small write stream by an order of magnitude over the RAID, and with a faster processor small write performance would be improved even more. The high CPU load in many of the performance measurements shows that even with hardware support, there are still significant demands on the file system CPU, and a fast processor is still required. Our current CPU was a bottleneck for small writes and concurrent operations.

In conclusion, Sawmill shows that combining a direct data path, a disk array, and a log-structured file system is a cost-effective method of providing high-bandwidth storage.

8. Acknowledgments

The authors would like to thank John Bongiovanni, Keith Bostic, and Drew Roselli for their comments on this paper. The assistance of the RAID group, in particular Ann Drapeau and Srinivas Seshan was instrumental in getting Sawmill running. This research was supported by the California MICRO program, ARPA grant N00600-93-C-2481, and NSF grant IRI-9116860.

9. References

- [CK91] Ann L. Chervenak and Randy H. Katz. Performance of a disk array prototype. In *Proceedings of the 1991 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 188–197, 1991.
- [CL91] L. Cabrera and D. Long. Exploiting multiple I/O streams to provide high data-rates. In *Proceedings of the 1991 USENIX Summer Conference*, June 1991.
- [CMS90] B. Collins, C. Mercier, and T. Stup. Mass-storage system advances at Los Alamos. In *Digest of Papers, Proc. Tenth IEEE Symposium on Mass Storage Systems*, pages 77–81, May 1990.

[dJKH93] Wiebran de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. The Logical Disk: A new approach to improving file systems. In *Proceedings of the Fourteenth SOSP, Operating Systems Review*, volume 27, pages 15–28, December 1993.

[DSH⁺94] Ann L. Drapeau, Ken Shirriff, John H. Hartman, Ethan L. Miller, Srinivasan Seshan, Randy H. Katz, Ken Lutz, David A. Patterson, Edward K. Lee, Peter M. Chen, and Garth A. Gibson. RAID-II: A high-bandwidth network file server. In *Proceedings of the 21st International Symposium on Computer Architecture*, Chicago, IL, April 1994. To appear.

[GPS93] G. A. Gibson, R. H. Patterson, and M. Satyanarayanan. A status report on research in transparent informed prefetching. *ACM Operating System Review*, 27(2):21–34, April 1993.

[HCF⁺90] C. Hogan, L. Cassell, J. Foglesong, J. Kordas, M. Nemanic, and G. Richmond. The Livermore Distributed Storage System: Requirements and overview. In *Digest of Papers, Proc. Tenth IEEE Symposium on Mass Storage Systems*, pages 6–17, May 1990.

[HO93] John H. Hartman and John K. Ousterhout. The Zebra striped network file system. In *Proceedings of the 14th Symposium on Operating System Principles*, pages 29–43, Asheville, NC, December 1993. ACM.

[IBM91] IBM. *0661 Functional Specification Model 371, Version 0.7*, February 18 1991.

[MJLF84] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[MK91] L. McVoy and S. Kielman. Extent-like performance from a UNIX file system. In *Proceedings of the 1991 USENIX Winter Conference*, 1991.

[MRK93] J. Menon, J. Roche, and J. Kasson. Floating parity and data disk arrays. *Journal of Parallel and Distributed Computing*, 17:129–139, 1993.

[Nel90] B. Nelson. An overview of functional multiprocessing for network servers. Technical report, Auspex Systems Inc., July 1990.

[PGK88] David Patterson, Garth Gibson, and Randy Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD Conference*, pages 109–116, June 1988.

[Ros92] Mendel Rosenblum. *The Design and Implementation of a Log-structured File System*. PhD the-

sis, U.C. Berkeley, June 1992. Report UCB/CSD 92/696.

[SBMS93] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In *1993 Winter Usenix*, pages 307–326, January 1993.

[SGH93] D. Stodolsky, G. Gibson, and M. Holland. Parity logging overcoming the small write problem in redundant disk arrays. *Computer Architecture News*, 2(2):64–75, May 1993.

[Wil90] D. Wilson. The Auspex NS5000 fileserver. *Unix Review*, 8(8):91–102, August 1990.

[Wil91] J. Wilkes. DataMesh - parallel storage systems for the 1990s. In *Digest of Papers, Proc. Eleventh IEEE Symposium on Mass Storage Systems*, pages 131–136, October 1991.

Author Information

Ken Shirriff is a graduate student in computer science at the University of California, Berkeley. He is a member of the Sprite network operating system project. His interests include distributed operating systems, computer graphics, and fractals. He received a B.Math. degree from the University of Waterloo in 1987, a M.S. in computer science from UC Berkeley in 1990, and expects to receive his Ph.D. in 1994. He can be reached at shirriff@cs.berkeley.edu.

John K. Ousterhout is a Professor in the Department of Electrical Engineering and Computer Sciences at the University of California at Berkeley. His interests include operating systems, distributed systems, user interfaces, and computer-aided design. He is currently leading the development of Sprite, a network operating system for high-performance workstations, and Tcl/Tk, a programming system for graphical user interfaces. Ousterhout is a recipient of the ACM Grace Murray Hopper Award, the National Science Foundation Presidential Young Investigator Award, the National Academy of Sciences Award for Initiatives in Research, the IEEE Browder J. Thompson Award, and the UCB Distinguished Teaching Award. He received a B.S. degree in Physics from Yale University in 1975 and a Ph.D. in Computer Science from Carnegie Mellon University in 1980.

NFS Version 3

Design and Implementation

Brian Pawlowski

Chet Juszczak

Peter Staubach

Carl Smith

Diane Lebel

David Hitz

Abstract

This paper describes a new version of the Network File System (NFS) that supports access to files larger than 4GB and increases sequential write throughput seven-fold when compared to unaccelerated NFS Version 2. NFS Version 3 maintains the stateless server design and simple crash recovery of NFS Version 2, and the philosophy of building a distributed file service from cooperating protocols. We describe the protocol and its implementation, and provide initial performance measurements. We then describe the implementation effort. Finally, we contrast this work with other distributed file systems and discuss future revisions of NFS.

1. Introduction

"It is common sense to take a method and try it. If it fails, admit it frankly and try another. But above all, try something." *Roosevelt, 1932*

The NFS protocol is a collection of remote procedures that allow a client to transparently access files stored on a server [Joy84a]. It is independent of architecture [RFC1014], operating system, network, and transport protocol. The protocol does not exactly match the semantics of any existing system. Instead, it provides a basis for portability and interoperability.

NFS Version 1 existed only within Sun Microsystems and was never released. NFS Version 2 was implemented in 1984 and released with SunOS 2.0, in 1985 [Sandberg85]. NFS Version 2 implementations exist for a variety of machines, from personal computers to supercomputers.

2. NFS Version 2 protocol problems

Several problems in NFS Version 2 could only be solved through a new version of the protocol. The 4GB file size limitation has recently become a pressing problem, although implementations of NFS on larger machines such as Cray supercomputers exposed this limitation years ago.

Performance suffers under NFS Version 2 because the protocol requires servers to write data and file system metadata to stable storage (usually disk) synchronously, before replying successfully to a client WRITE request [Ousterhout90]. The performance problem with synchronous writes was recognized early. NFS Version 2 has an artifact of a proposed interface for asynchronous writes (the undefined WRITECACHE procedure).

Implementations have attacked this problem in several ways. [Moran90] describes the Prestoserve product, which interposes a software driver between the file system and disk driver to accelerate writes by using nonvolatile RAM. [Juszczak94] describes a technique called *write gathering*, which exploits the tendency of more-capable clients to send write requests in clusters to gain parallelism. The author implemented a server that gathers several writes before synchronously committing the data to disk, thereby amortizing the cost of synchronous writes over several requests. [Hitz94] describes an integrated file server design that combines a log-based file system and non-volatile RAM to solve the synchronous write bottleneck.

Some implementations provide an "unsafe" option in their NFS Version 2 server that disables committing modified data to stable storage. While improving performance, this option violates the stable storage guarantee in the NFS Version 2 protocol and can result in data loss. This option has resulted in heated debate.

Lack of consistency guarantees was cited as the cause of excessive requests over-the-wire resulting in increased server loading and response time [Howard88]. [Reid90] and [Arnold91] describe additional problems with NFS Version 2.

3. The NFS Version 3 protocol

Engineers from several companies gathered for a two-week series of meetings in July, 1992, in Boston, MA. to develop an NFS Version 3 specification. The

group's goal was to address compelling issues in the current protocol that could not be solved by implementation practice. The only absolute requirement was 64-bit file size support.

Other issues under consideration included the following:

- Solving the write throughput bottleneck
- Minimizing the work needed to create an NFS Version 3 implementation given an existing NFS Version 2 implementation
- Ensuring that implementation of the new protocol is feasible on less-capable client operating systems (for example, DOS)
- Completely documenting the resulting protocol and annotating it with implementation examples to aid developers
- Deferring new features to subsequent revisions of NFS due to time constraints

Above all, the driving principles were the following:

- Keep it simple
- Get it done in a year
- Avoid anything controversial

Although it wasn't an absolute requirement, we felt that solving the write throughput bottleneck would provide the most compelling feature.

3.1. Changes introduced

NFS Version 3 represents an evolution of the existing NFS Version 2 protocol. Most of the original design features described in [Joy84a], [Sandberg85], and [RFC1094] persist. This revision introduces the following major changes:

- Sizes and offsets are widened from 32 bits to 64 bits.
- The WRITE and COMMIT procedures allow reliable asynchronous writes.
- A new ACCESS procedure fixes known problems with super-user permission mapping and allows servers to return file access permission errors to the client at file open time to provide better support for systems with Access Control Lists (ACLs).
- All operations now return attributes to reduce the number of subsequent GETATTR procedure calls.
- The 8KB data size limitation on the READ and WRITE procedures is relaxed.
- A new REaddirplus procedure returns both file handle and attributes to eliminate LOOKUP calls when scanning a directory.
- File handles are of variable length, up to 64 bytes, as needed by some implementations

[Pawlowski89]. (We kept the file handle size small enough to allow efficient DOS implementations.)

- Exclusive CREATE requests are supported.
- File names and path names are now specified as strings of variable length, with the maximum length negotiated between the client and server (with the PATHCONF [POSIX90] procedure).
- The errors the server can return are enumerated in the specification—no others are allowed.
- The notion of blocks is discarded in favor of bytes.
- The new NFS3ERR_JUKEBOX error informs clients that a file is currently off-line and that they should try again later.

Appendix 1 provides a summary of the protocol differences between NFS Version 2 and NFS Version 3. Refer to [NFS3] for more details.

At least eight new versions of NFS have been proposed to fix NFS Version 2, none of which has ever been completely implemented. Public reviews of the draft versions of new protocol specifications have occurred continuously since early 1987. Several changes included in NFS Version 3 first appeared in those eight drafts.

3.2. What was avoided

"Let joy and innocence prevail." *Toys, 1993*

In the years since the NFS protocol was first described, implementation practice solved several problems originally thought to require a protocol revision, although minor, undocumented changes were made to the protocol without a formal revision. In practice, NFS Version 2 mostly works, and we tried not to break it. Accepting common implementation practice reduced the number of changes needed to produce NFS Version 3. Minor protocol changes were cleaned up and incorporated into this work.

We decided to maintain the current stateless design of NFS and not include strict cache consistency. When we defined NFS Version 3, research work on consistent versions of NFS was incomplete. Delaying support for 64-bit file sizes to explore adding stateful consistency was unacceptable. In addition, it seemed clear that supporting strict data consistency introduces complexities that would preclude implementation on less-capable clients. Finally, the recovery benefits of a stateless server were clear, while the issues of stateful recovery were not.

The stateless server design of NFS creates a problem with the replaying of nonidempotent requests. An idempotent request such as LOOKUP can be successful-

ly executed any number of times. A nonidempotent request such as REMOVE can be successfully executed only once. Primarily a correctness problem, this condition has been solved through the use of a reply cache of recently serviced requests on the server [Juszczak89]. Proposed protocol extensions to NFS attempted to fix this but were essentially misguided. The Boston group simply acknowledged the effectiveness of this implementation technique and left the protocol alone.

Many other changes to NFS Version 2 were proposed in the eight protocol revisions, including the following:

- The ZERO procedure to punch holes in a file
- Append mode writes
- Record-oriented I/O support
- File name to include versions
- User and group fields as strings
- Extended attributes (arbitrary key/value pairs)
- Well-defined UID mapping procedures
- Advisory close procedure
- Resource fork support for the Macintosh
- Multiple OS-dependent name spaces
- A get server statistics procedure

Most of the above proposed features were rejected because by 1992 implementers had worked around purported "protocol limitations" that would prevent implementations on non-UNIX platforms. Other proposed features above were rejected because they were specific to a single operating system. The remaining proposed features were discarded because they attempted to solve a problem simplistically that was best solved correctly (for example, append mode writes versus a full consistency protocol).

4. Design and implementation

NFS Version 3 defines a revision to NFS Version 2; it does not provide a new model for distributed file systems. Because of this, NFS Version 3 resembles NFS Version 2 in design assumptions, file system and consistency model, and method of recovering from server crashes. For a general description of the implementation issues of NFS, see [Sandberg85], [Israel89], [Juszczak89], [Pawlowski89], [Macklem91], and [Juszczak94].

4.1. NFS design

NFS achieves architecture and operating system independence through a strict separation of the protocol and its implementation. The protocol is the interface by which clients access files on a server. A client or server implements the protocol by mapping local file

system actions into the file system model defined by NFS. The NFS protocol does not dictate how a server implements the interface or how a client should use the interface [Satyanarayanan89]. For example, the NFS Version 3 protocol does not define how a client should manage cached data, but it does provide information to improve cache management.

Although implementations have been used to illustrate aspects of the NFS protocol, the specification itself is the final description of how clients should access servers. Semantic details that were not fully described in the NFS Version 2 specification [RFC1094] have proven, in practice, not to be a problem and have been worked out through interoperability testing. Most problems are flaws in implementations, instead of the protocol design.

The NFS protocol is stateless; that is, each request contains sufficient information to be completely processed without regard to other requests. The server does not need to maintain state about any previous requests¹ other than file data on stable storage, and a map of file handles (opaque tokens used by clients to access files) to files derived from file system data. Of course, most servers cache file data that has been synchronized to disk to improve performance. However, this cached data is not needed for correct operation.

Server crash recovery is simple. A client need only retry a request until the server responds; the client does not know that the server has rebooted (although the user may notice delayed responses). Experience at Sun with *network disk* (*nd*), an earlier method of sharing disk storage on a network, led to the stateless server requirement in the initial design of NFS [Joy84b].

The NFS Version 3 protocol requires that modified data on the server be flushed to stable storage before replying. Only asynchronous writes are accepted. NFS clients block on *close(2)* until all data is flushed to stable storage on the server, to return any errors to the application that might occur during delayed writes (for example, out of space).

NFS clients are decidedly not stateless. NFS clients hold modified data that has not been flushed to the server as well as cache file handles and attributes. Clients typically use attribute information, such as file modification time, to validate cached information. When a client crashes no recovery is necessary for either the client or the server.

¹ To be precise, the reply cache on a server contains volatile state needed for correctness [Kazar94]. See [Bhide91] for further discussion on the reply cache and its implications for server correctness. TCP-based implementations of NFS still need a reply cache to prevent destructive replay following connection re-establishment.

Thus, NFS servers are stupid and NFS clients are smart. NFS Version 3 offers the possibility of potentially smarter clients.

4.2. Multiple version support

The Remote Procedure Call (RPC) protocol provides explicit support for multiple versions of a service [RFC1057]. The client and server implementations of NFS Version 3 provide backward compatibility with NFS Version 2 by supporting both NFS Version 2 and NFS Version 3. By default, an RPC client and server bind using the highest version number they both support. Client or server implementations that cannot support both versions (for example, due to memory restrictions) should support NFS Version 2.

4.3. Implementation issues

A primary goal in restricting the changes between NFS Version 2 and NFS Version 3 was to minimize new implementation issues. Implementation issues exist in the following areas:

- 64-bit file sizes and offsets
- Asynchronous writes
- READDIRPLUS—read directory with attributes
- NFS3ERR_JUKEBOX
- Weak cache consistency
- Other issues

4.3.1 64-bit file sizes and offsets

The 64-bit extensions in NFS Version 3 introduce problems with mismatched clients and servers, such as a 32-bit client and a 64-bit server, or a 64-bit client and a 32-bit server.

A 64-bit client will never encounter a file that it cannot handle when using a 32-bit server. If it sends a request that the server cannot handle, the server should return NFS3ERR_FBIG.

The problems posed by a 32-bit client and a 64-bit server are more difficult. The server can handle anything that the client can generate. However, the client cannot handle a file whose size can not be expressed in 32-bits, and will not properly decode the size of the file into its local attributes structure. One solution is for the client to deny access to any file whose size cannot be expressed in 32 bits. This introduces anomalous behavior when a file is extended by the client beyond its limit, thus rendering the file inaccessible.

Another solution is for the client to map any size greater than it can handle to the maximum size that it can handle, effectively “lying” to the application program. This allows the application access to as much of the file as possible given the 32-bit offset restriction.

Although this solution eliminates the anomalous behavior described in the first solution, it introduces the problem that a client might be able to access only part of a file. However, other solutions exist.

4.3.2 Asynchronous writes

NFS Version 3 asynchronous writes eliminate the synchronous write bottleneck in NFS Version 2. When a server receives an asynchronous WRITE request, it is permitted to reply to the client immediately. Later, the client sends a COMMIT request to verify that the data has reached stable storage; the server must not reply to the COMMIT until it safely stores the data.

Asynchronous writes as defined in NFS Version 3 are most effective for large files. A client can send many WRITE requests, and then send a single COMMIT to flush the entire file to disk when it closes the file. This allows the server to do a single large write, which most file systems handle much more efficiently than a series of small writes. For very large files, the server can flush data in the background so that most of it will already be on disk when the COMMIT request arrives.

Asynchronous writes are optional in NFS Version 3, and specific client or server implementations can choose not to support this feature. A server can choose to flush asynchronous write requests to stable storage. In this case, the server indicates this in the WRITE reply. Clients with insufficient memory to support the necessary buffering required for server crash recovery can always request synchronous writes.

4.3.2.1 Crash recovery

The design of asynchronous writes is consistent with the stupid server and smart client philosophy of NFS. The client is required to keep a copy of all uncommitted data to support recovery following a server crash. The replies for WRITE and COMMIT requests include a *write verifier* that clients use to detect server crashes. The write verifier is an 8-byte value that the server must change whenever it crashes. Servers commonly use their boot time as a write verifier, because it is guaranteed to be unique after each crash. The client must save the write verifier returned by each asynchronous WRITE request and compare it to the write verifier returned by a later COMMIT request. If the write verifiers do not match, then the client assumes that the server has crashed and rebooted.

The client must then rewrite all uncommitted data. Clients can push data with synchronous writes following server failure. The client can delay rewriting data when it detects a crash to avoid flooding a newly rebooted server with WRITE requests. Figure 1 shows

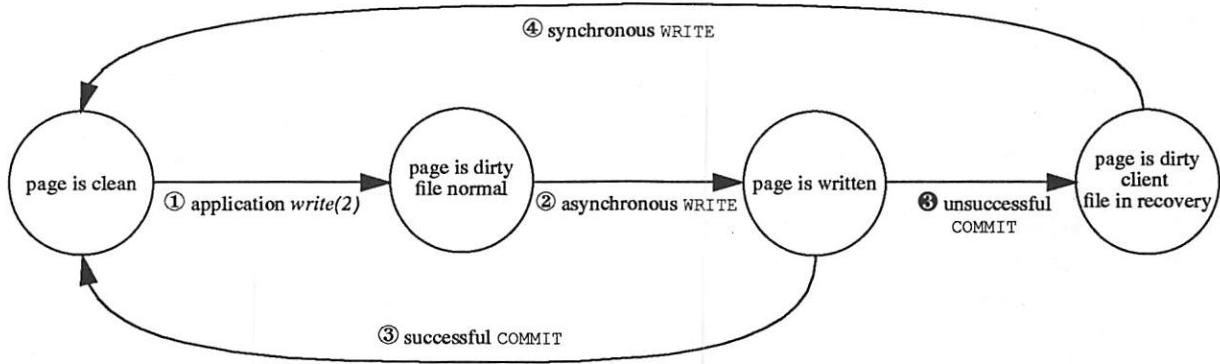


Figure 1. Client page states with asynchronous writes (The Digital OSF/1 implementation). This diagram shows the state changes that occur as a page of memory containing file data is modified, written, and then committed. ① A local application modifies the page, and it is marked dirty. ② The client asynchronously writes the data to the server. The client stores the write verifier from the asynchronous write request with each page. An explicit `msync(2)`, `fsync(2)` or `close(2)` from the application, a file system sync, or page reclamation will trigger a COMMIT. The write verifier returned from the COMMIT request is compared against those stored with the written pages. ③ The page's write verifier matches the returned verifier, and the commit succeeds. ④ The write verifier for the page does not match the returned write verifier, triggering recovery. ④ The client synchronously writes the data to the server.

the state changes that occur as a page of memory containing file data is modified, written to the server, and then committed.

4.3.2.2 Server details

An NFS Version 3 server makes the following three guarantees:

- For a synchronous WRITE request, the server will commit to stable storage all data and modified metadata.
- The server will not discard uncommitted data without changing the write verifier.
- The server will commit the file's data and modified metadata to stable storage for the range specified in the COMMIT request before reporting success.

Other conditions arise in which the write verifier must change. For example, the server must change the write verifier on failover if NFS Version 3 forms the basis of a non-shared memory, highly available implementation of NFS [Bhide91]. The unsynchronized data is not available to the backup processor, and there is no guarantee that the primary processor was able to flush uncommitted data to stable storage before going down.

If a server is shut down cleanly, it could be advantageous to save the write verifier for reuse when the server is brought back on line. This avoids triggering client rewrites of already committed data.

4.3.2.3 Data sharing

Asynchronous writes make write sharing without using a higher-level application synchronization protocol even less attractive than with NFS Version 2. NFS Version 3 clients preserve close-to-open consistency: clients typically block on a `close(2)` until all data is flushed to server stable storage and revalidate cached data with an attribute check on `open(2)`. Strictly speaking, close-to-open consistency is only an implementation practice. Data sharing semantics of NFS Version 3 differ from those of NFS Version 2 if an NFS Version 3 server reboots and loses uncommitted data. Because write sharing between NFS Version 2 clients was never supported in the absence of locking, changes in essentially undefined behavior is not considered a major issue.

4.3.3 REaddirplus

NFS Version 3 contains a new operation called REaddirplus, which returns file handles and attributes in addition to the directory information returned by REaddir.

REaddirplus exploits observed request sequences generated by NFS Version 2 clients. For example, when a UNIX user types “`ls -F dir`” to browse a directory containing 20 entries, the `ls` command opens the target directory, reads it, and then calls `stat(2)` 20 times. In NFS Version 2, a REaddir request would be followed by 20 sequential LOOKUP requests to retrieve attributes (and file handles). In NFS Version 3, a single REaddirplus retrieves the name list and attributes for the 20 entries, significantly reducing the command execution time.

There are some drawbacks to REaddirplus, however. A REaddirplus is more expensive than a corresponding REaddir. Results from an implementation that generates exclusively REaddirplus requests show a performance drop because attributes for all entries in a directory are fetched repeatedly for every access to a directory.

The REaddirplus operation can be viewed as a way to get the contents of a directory and to populate name and attribute caches for the entries in that directory at the same time. The REaddirplus operation should be used only when reading a directory for the first time or when rereading a directory whose cache entry was invalidated because the directory was modified. A REaddirplus should not be issued when a valid cache entry for a directory exists, because it is likely that a REaddirplus operation was recently issued to populate the various caches with directory entry attributes and file handles.

4.3.4 NFS3ERR_JUKEBOX

NFS3ERR_JUKEBOX² lets servers inform clients that a file is temporarily inaccessible (archived offline or locked against modification for backup) and that they should retry the request later. It is intended to improve the behavior of NFS in hierarchical storage management applications.

In NFS Version 2, a server performs one of three actions if a file is temporarily inaccessible. The first action is to drop the request, which forces the client into normal back-off and retransmission. The request will be satisfied at some later time on a retry. The second action is to have the server block a service thread until the file again becomes accessible. The second action is often implemented inadvertently; because clients employ mechanisms like biods to gain parallelism and will emit several related requests to one file, blocking server threads can hang the server. The third action is to return some error to the client, thus rejecting the request.

An NFS Version 3 server returns NFS3ERR_JUKEBOX when a file is temporarily inaccessible. The client operating system does not return the error to the application but handles it internally by aggressively delaying reissue of the request, thereby reducing server load due to request retransmission. After a tunable delay, the request is reissued. The client

²The term "JUKEBOX" is a long standing joke in the NFS community. We kept the historical error name even though it incorrectly implies a binding to a particular HSM mechanism. Given the generic intent of the error, NFS3ERR_TMP_INACCESSIBLE would be more appropriate.

should reissue the request with another transmission id.

4.3.5 Weak cache consistency

Many NFS Version 2 clients cache file and directory data to improve performance. To determine whether cached data is valid, a client sends a GETATTR request. If the new modification time from the server matches the modification time in the client's cached attributes, then the client assumes its cache is up-to-date. If the modification times don't match, then the file must have changed, and the client invalidates its cache.

This method fails when the client itself modifies the file being cached. For example, if a client writes to one part of a file, cached data for other parts is probably still valid. But it is impossible for the client to be sure, because the client's own WRITE request updated the file's modification time. A reckless client might keep the cache data (which is dangerous), and a cautious client might invalidate the cache (which is slow).

Weak cache consistency offers an alternative by helping clients determine more accurately when to invalidate their cache. The reply for each NFS Version 3 request that can modify data includes two versions of the file's attributes: pre-operation attributes from just before the server performed the operation and post-operation attributes from just after the operation. If the modification time in the pre-operation attributes from the server matches the cached attributes on the client, then the client's cache is valid. The client should update its attribute cache with the new post-operation attributes.

Weak cache consistency does not provide true consistency such as found in Sprite [Nelson88]. With weak cache consistency, clients might see an inconsistent view of server data. For example, one client might have modified a file locally but not yet flushed the new data to the server. Even if it has, a second client will only verify modification times when a file is first opened or when the cached attributes time out. As a result, a second client's cache may be out of date.

Some servers may be unable to generate pre-operation attributes, so clients should be prepared to fall back to NFS Version 2 behavior. Since weak cache consistency is just a hint, client implementations are free to use it or ignore it.

4.3.6 Other issues

Two changes in NFS Version 3 impose extra work on the client. For many NFS Version 3 requests, it is optional to return file handle and attribute information that is mandatory in NFS Version 2. For example, in

NFS Version 2, the CREATE request must return the file handle and attributes for the newly created file, but in NFS Version 3, their return is optional. As a result, an NFS Version 3 client must be prepared to issue a LOOKUP after each CREATE, in case the server does not return a file handle for the new file. Furthermore, in NFS Version 3, it is optional for LOOKUP to return attributes, so the client must also be prepared to issue a GETATTR.

NFS Version 2 servers are required to accept all or none of the data in a WRITE request. In NFS Version 3, a server can accept only some of the data in a write, and the client is expected to send the rest a second time. For example, a client might send an 8192 byte request, but a server might choose to accept only 1 byte. The client must be prepared to send the remaining 8191 bytes a second time, and again, the server might choose not to accept the entire request.

In practice, these features are unlikely to be a problem because most server implementations will always return optional information and accept the entire contents of WRITE requests.

4.4. Changes to related protocols

NFS Version 3 continues the philosophy of building a network file service from a collection of cooperating protocols. The mount protocol (MOUNT) allows an NFS client to gain access to an exported directory on a server, and the network lock manager protocol (NLM) supports remote file locking for NFS.

Changes to the file handle and file size fields in NFS Version 3 required corresponding changes in MOUNT and NLM, so new versions of both protocols have been released. The new MOUNT specification allows a successful mount to return a list of acceptable RPC authentication flavors (such as DES or Kerberos) for the client to use. Automounter facilities can use this information to correctly access servers which require certain flavors of authentication. The new MOUNT protocol is also slightly cleaner than the previous one. For example, legal error values have been enumerated instead of allowing any UNIX error number.

5. Performance

A major goal of NFS Version 3 was to improve performance, especially in write throughput. Performance was improved by the following:

- Providing reliable asynchronous writes
- Removing the 8KB data size limitation for READ and WRITE requests
- Providing a REaddirplus procedure that returns

file handles and attributes with directory names

- Returning attribute information in all replies
- Providing weak cache consistency data to allow a client to more effectively manage its caches

5.1. Test setup

We measured Digital's OSF/1 implementation of NFS Versions 2 and 3. The local file system employed for these tests was the Berkeley Fast File System with enhanced clustering [McVoy91]. Except where noted, the following configuration was used to generate the performance results:

- Two Digital Model 3000/600 96MB workstations
- Private FDDI network
- Server running 16 nfsds (multiple threads of execution used on an NFS server to gain parallelism).
- Client running 7 nfsiods (or biods—multiple threads of execution used on an NFS client to gain parallelism)
- With and without Prestoserve on server, using 1MB NVRAM
- With and without write gathering on server
- Server configured with one 1GB RZ26 SCSI disk, 2.3 MB/sec raw transfer rate.

The tests ran with NFS running on top of UDP with a maximum transfer size of 8KB. The larger transfer sizes permitted by NFS Version 3 were not exploited. Measurements at SunSoft on a system using larger than 8KB transfer sizes showed improved write throughput, presumably from the reduced file system overhead resulting from fewer separate I/O requests and fewer RPC messages over-the-wire.

5.2. Sequential write throughput

Figure 2 shows the results of writing a 10MB file over a private FDDI network using NFS Version 2 and NFS Version 3 protocols and varying the server configuration to enable/disable Prestoserve acceleration and server write gathering. We consider the NFS Version 2, no write gathering, no Prestoserve configuration to be the average NFS write throughput available today. We believe that the NFS Version 2, write gathering, Prestoserve configuration provides competitive NFS write throughput. We observe the following:

- NFS Version 2 with Prestoserve and NFS Version 3 delivers the maximum raw device rate to the remote client.
- NFS Version 3 with asynchronous writes at 2323 KB/s delivers only 1% less throughput than NFS Version 2 with Prestoserve and write gathering at 2346 KB/s, but it consumes 36% less server CPU.

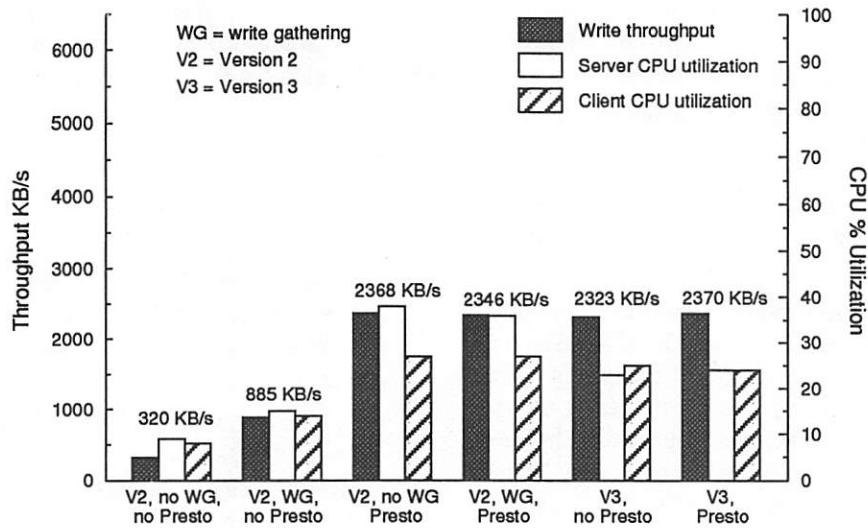


Figure 2. Comparisons of 10MB file writes over FDDI, Digital OSF/1, Digital 3000/600

- At 2323 KB/s, NFS Version 3 is seven times faster than NFS Version 2 at 320 KB/s for a typical configuration with no write gathering and no Prestoserve.

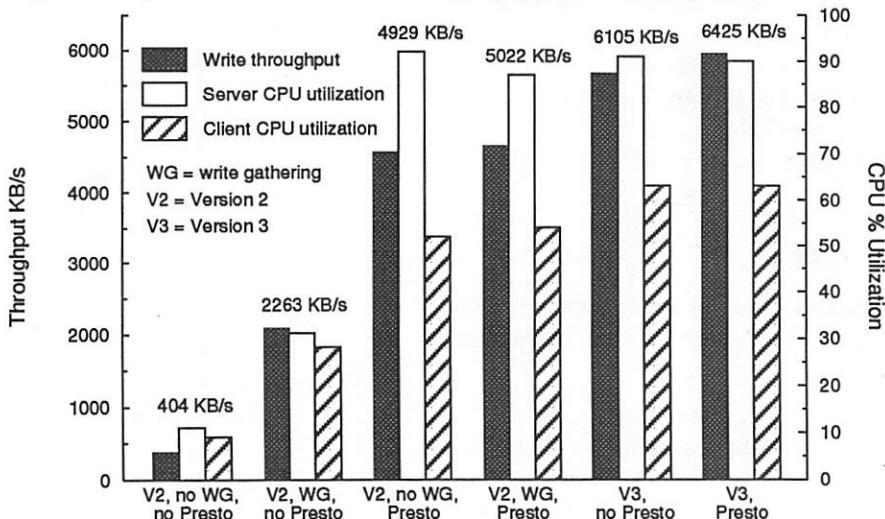
The NFS Version 3 client emitted only asynchronous writes in these tests; therefore, server write gathering had no effect. This configuration is not shown. Prestoserve further improves NFS Version 3 asynchronous writes because there is a synchronous component to writing metadata during local file system clustering. NFS Version 2 with Prestoserve provides higher throughput on a single disk system than NFS Version 3, because Prestoserve masks the cost of reduced cluster transfer sizes and missed rotations seen in its absence. Multiple spindles can help mask these effects in the absence of accelerator hardware.

It was clear that the disk was the bottleneck for the above test, given the low CPU utilizations, available network bandwidth on FDDI (100 Mbit/s), and the raw speed of the disk. To remove the disk bottleneck, we made a second set of runs, sequentially writing a 40MB file, with the following configuration changes:

- Server configured with four 2GB RZ28 SCSI disks, each 4.8 MB/sec raw transfer rate, four-way striped
- Client running 15 nfsiods (or biods)

The results in Figure 3 show that for sufficiently large files on a non-disk bound server, NFS Version 3 delivered 6105 KB/s, compared to an NFS Version 2 server with Prestoserve and write gathering that delivered 5022 KB/s. NFS Version 3 delivered 22% more throughput at a similar server CPU utilization. The

Figure 3. Comparisons of 40MB file write over FDDI, Digital OSF/1, Digital 3000/600



Test	Basic test description	Version 2 NFS no Presto	Version 3NFS no Presto	Version 2 NFS with Presto	Version 3 NFS with Presto
1	File and directory creation create 155 files 62 directories 5 levels deep	8.39	8.21	0.87	0.77
2	File and directory removal remove 155 files 62 directories 5 levels deep	3.71	3.66	1.60	1.20
3	lookups across mount point 500 getwd and stat calls	0.81	0.81	0.74	0.71
4	setattr, getattr, and lookup 1000 chmods and stats on 10 files	11.18	11.18	0.94	1.00
5a	write 1MB file 10 times throughput	12.00 869 KB/s	5.35 1957 KB/s	4.68 2238 KB/s	4.69 2234 KB/s
5b	read 1MB file 10 times throughput	1.48 7056 KB/s	1.48 7052 KB/s	1.49 7019 KB/s	1.47 7128 KB/s
6	readdir 20500 entries read, 200 files	2.87	2.79	1.40	0.96
7	link and rename 200 renames and links on 10 files	6.71	6.71	1.22	1.10
8	symlink and readlink 400 symlinks and readlinks on 10 files	6.73	6.70	1.25	0.98
9	statfs 1500 statfs calls	0.92	1.50	0.92	1.40
Basic tests NFS RPC count		13166	11032	13166	11032
Total NFS RPC count for Basic, General and Special tests		21865	17764	21865	17764

Table 1: Connectathon Basic test suite results, 7 boids, single disk spindle, (results in seconds, except as noted)

maximum throughput of 6425 KB/s was achieved with NFS Version 3 and Prestoserve. Throughput increased with this configuration change, but not to the point of the disk bandwidth limitation or CPU exhaustion. The bottleneck moved to the network because of the limited number of stations, limited application parallelism, and FDDI token holding time characteristics of the network interfaces.

We conclude that asynchronous writes improve both client throughput and server efficiency. They provide most of the benefits associated with running an NFS Version 2 server in “unsafe” mode, while ensuring data reliability after server failure³. Prestoserve should still accelerate small file writes, as well as other modifying requests like CREATE and REMOVE.

5.3. Connectathon test suite results

Because the LADDIS benchmark generates NFS Version 2 RPC calls directly to measure server performance [Wittle93], it cannot measure NFS Version 3 without modification. As an alternative, we ran the Connectathon test suite, which was developed to test the interoperability of NFS implementations. It runs

on the client on a remotely mounted directory and exercises both client and server NFS code. It consists of three passes that cover the basic functionality of a file system. The Basic pass isolates specific features of the client file system, and consists of ten separate tests. Testing a single client file system feature typically generates a mix of NFS requests. The General pass runs multiple simultaneous large compiles, as well as *nroff(1)*. The Special pass exercises boundary cases in NFS operations.

Table 1 contains the results of running the Connectathon test suite. We conclude the following from these results:

- Again, NFS Version 3 asynchronous writes are clearly a win (see test 5a).
- Prestoserve remains useful on the server for other metadata operations (CREATE, REMOVE, etc.), as shown by tests 1, 2, 4, 6, 7 and 8. Test 6 performs file deletions in addition to reading directory entries, which explains the improvement with Prestoserve.
- NFS Version 3 reduces the total number of RPC messages by 18% compared to NFS Version 2. The reduction is due entirely to the increased frequency of returned attributes and better cache management through weak cache consistency data. This reduction more than offsets the calls to the new ACCESS and COMMIT RPC procedures.

³ [Nelson88b] suggests that unsafe writes would provide greater throughput than asynchronous writes with close-to-open consistency. That is, assuming that COMMIT blocks until all remaining data is on disk when a file is closed, unsafe mode implementations which do not block would clearly perform better. For large files, this effect should be minimal.

```

NFS Version 2
calls
21865
null      getattr    setattr   root      lookup    readlink   read
0 0%     4058 18%  1168 5%  0 0%    6954 31%  250 1%   1779 8%
wrcache   write      create    remove   rename    link      symlink
0 0%     1881 8%   675 3%  1175 5%  352 1%   250 1%   250 1%
mkdir    rmdir     readdir  statfs   173 0%   972 4%  1755 8%
173 0%   173 0%   972 4%  1755 8%

NFS Version 3
calls
17764
null      getattr    setattr   lookup    access    readlink   read
0 0%     1282 7%   1168 6%  5499 30%  309 1%   250 1%   1731 9%
write    create    mkdir    symlink  mknod    remove    rmdir
1881 10% 675 3%  173 0%  250 1%   0 0%    1175 6%  173 0%
rename   link      readdir  readdir+ fsstat   fsinfo    pathconf commit
352 1%   250 1%   758 4%  18 0%   1755 9%  0 0%    0 0%   65 0%

```

Figure 4. Detail of RPC counts for all three passes of the Connectathon Test Suite

The read throughput results from test 5b reflect over-the-wire data transfers. Test 5b was modified to use the *mmap(2)* system call to invalidate the client's data cache, forcing the requests to go over-the-wire. However, the data was cached on the server. The detailed RPC counts for the NFS Version 2 and Version 3 results are shown in Figure 4.

5.4. *find(1)* results

The *find(1)* command was used to measure the effect of REaddirplus. *find(1)* scanned a remote file tree containing 9612 files distributed over 155 directories that were up to seven levels deep. The results are shown in Table 2. The over-the-wire byte counts include all protocol headers.

Using REaddirplus to fetch file handles and attributes of entries in a directory reduces the *find(1)* execution time by 36%, compared to NFS Version 2. Re-

duced execution time can be attributed primarily to the tenfold reduction in over-the-wire messages. The 155 GETATTR requests are generated to ensure close-to-open consistency when opening a directory. Using the REaddirplus procedure in NFS Version 3 reduced the total bytes transferred over-the-wire by 43% and the cumulative server CPU (percent utilization \times elapsed time) by 46%, compared to using the REaddir and LOOKUP procedures in NFS Version 2. REaddirplus is clearly a win in this example.

The test was rerun with REaddirplus disabled in NFS Version 3. The last column in Table 2 shows these results. Disabling REaddirplus increases execution time by 95%, compared to the NFS Version 3 result with REaddirplus enabled. More disturbing, execution time increased by 24%, compared to the NFS Version 2 results. We attribute this to the new ACCESS procedure and to larger message sizes in NFS Version 3, which increased the total bytes transferred

Table 2: *find(1)* results

	NFS Version 2	NFS Version 3 w/ REaddirplus	NFS Version 3 w/o REaddirplus
real time	9.9s	6.3s	12.3s
client system time	4.8s	3.1s	4.6s
GETATTR count	155 1%	155 13%	155 1%
LOOKUP count	10076 95%	155 13%	10076 92%
ACCESS count	n/a	310 27%	310 2%
REaddir count	173 1%	0 0%	181 1%
REaddirplus count	n/a	358 31%	0 0%
STATFS/FSSTAT count	155 1%	155 13%	155 1%
Total count	10559 100%	1133 100%	10877 100%
bytes sent	2108523	225209	2214575
bytes received	1973952	2088284	3270824
total bytes over the wire	4082475	2313493	5485399
client CPU utilization	53%	42%	44%
server CPU utilization	37%	32%	31%

by 34% when compared to NFS Version 2. Message sizes increased because new fields were added and old fields were widened.

This result illustrates a fundamental tradeoff in the NFS Version 3 design: increased RPC request and reply sizes are to be offset by new features in the protocol. Naive implementations that fail to use the new features will perform worse for some benchmarks than NFS Version 2, but effective use of new features will increase overall performance.

6. Cost of porting

The Digital OSF/1 implementation illustrates the effort and cost to port the SunSoft NFS Version 3 reference source into an existing Version 2 implementation. The source code size of an implementation that supports both protocols is roughly 30,000 lines (C code + comments + white space). The Version 2 and Version 3 specific portions of the total are about 12,000 lines each, with 6,000 lines of shared subroutines. Assuming engineers familiar with NFS Version 2, the effort needed to produce an implementation that supports both versions of the NFS protocol for initial testing is the following:

server	1 person-month
client (excluding asynchronous writes)	2 person-months
client asynchronous writes	1 person-month

Digital's OSF/1 based kernel uses a unified page cache managed by the virtual memory subsystem for both program text and file data. This complicated the client implementation of asynchronous writes because of dependencies on data structures and interfaces in the virtual memory system.

7. Related work

"Look on my works, ye Mighty, and despair!"
Ozymandias, Shelley, 1817

The NFS Version 3 protocol mitigates the need for NFS-specific write gathering techniques on clients that support asynchronous writes, because a server can now simply process clusters of related asynchronous writes as part of its local buffered file system activity [McVoy91]. However, NFS-specific write-gathering on servers is still useful in supporting less-capable NFS Version 3 clients that do not support asynchronous writes or more-capable clients that resort to synchronous behavior during recovery. The stable storage semantics for metadata modifying operations, such as CREATE, remain unaffected by NFS Version 3. Thus, a server can still benefit from fast stable storage. To a lesser extent, fast stable storage techniques still im-

prove asynchronous WRITE performance, especially for small files.

Adaptive retransmission strategies to improve the behavior of NFS over UDP (as described in [Nowicki89], derived from [Jacobson88]) and the use of TCP to improve performance over wide area networks [Macklem91], are applicable to NFS Version 3. NFS Version 3 relaxes the 8KB limitation on the data portion of a READ or WRITE request, permitting more efficient use of TCP.

Three efforts to revise the NFS protocol are related to this work. The first is Spritely NFS, described in [Srinivasan89], [Mogul92], and [Mogul93]. Spritely NFS uses a stateful server that controls client caching behavior to ensure consistency. State recovery following a crash is server-driven. The server keeps a nonvolatile list of old clients that are contacted during a grace period following reboot to initiate the rebuilding of state on the server. Spritely NFS employs consistency to address performance issues in NFS Version 2 by allowing clients to defer writes and by eliminating the need for clients to poll the server to detect file changes.

The second effort is NQNFS [Macklem94], which defines extensions to NFS Version 2 that are similar to those found in NFS Version 3. Size and offset fields were widened to 64 bits, and a REaddirplus procedure was added. Time-based leases provide a mechanism for data consistency and cache coherence among clients. Clients need to anticipate lease expiration. Clients do not have special recovery code. Instead, leases are short enough to expire while the server is rebooting, forcing clients to request renewals (thereby driving recovery) from the newly rebooted server. On reboot, a server accepts only writes during a grace period, after which it will grant new leases.

While the results of both NQNFS and Spritely NFS looked promising at the time we defined NFS Version 3, both were unfinished. We decided that adding consistency to NFS was contrary to our minimalist goals and best left for a subsequent revision.

The third effort, [Fadden92] and [Glover92], described Trusted NFS (TNFS), which defines a method for handling ACLs and data labels that conserves space. Acknowledging that security data can be large, TNFS maps the data into opaque tokens and requires a separate token mapping service to convert to and from a canonical over-the-wire format. We decided not to incorporate this work into NFS Version 3 because of instability in the POSIX ACL specification and the relative immaturity of extant implementations of TNFS.

DCE DFS [Kazar90] is related to NFS Version 3 only in that it describes an amount of effort that we

clearly did not want to undertake. Our primary goals were to improve NFS Version 2 and deploy a new version quickly. We preferred to retain the ease of server crash recovery, at the expense of not supporting some of the more valuable features of DCE DFS.

8. Future work

The strategy for using REaddirplus needs further research. Reading the contents of a very large directory with REaddirplus can eject potentially more valuable entries from client caches. Finding heuristics to guide choosing between REaddir and REaddirplus is hard because an NFS client cannot tell whether an application will need attribute information for a directory's children or not. More experience could lead to better heuristics than the simple ones used now.

An NFS Version 3 client trying to do effective cache management with weak cache consistency requires that the server guarantee atomicity of modifying operations and pre- and post-operation attribute generation. The performance cost of supporting such atomicity on the server is not fully understood, particularly for multiprocessor server implementations where extensive locking could result in unwanted serialization. More analysis is needed. Weak cache consistency with the WRITE procedure provides no useful sharing semantic.

Additional characterization and tuning of NFS Version 3 under more complex workloads is needed. An NFS Version 3 LADDIS benchmark is needed. Tuning NFS Version 3 implementations should not pose insurmountable problems.

We did not expect the NFS Version 3 specification to be perfect. Our hope is that the protocol specification will grow to reflect common practice and provide guidelines on conforming behavior. The development of an NFS Version 3 Validation Suite by SunSoft will aid interoperability. Finally, interoperability testing of implementations at Connectathon remains the cornerstone of successful file sharing with NFS.

8.1. NFS Version 4

In defining NFS Version 3, we assumed that other protocol revisions would follow, allowing us to defer features. Improved data and cache consistency is an obvious candidate for NFS Version 4. POSIX write-sharing semantics exist today on a single NFS client. NFS Versions 2 and 3 support a client-driven bounded time-based model for write sharing [Kazar88], with close-to-open consistency. This model does not provide sufficient guarantees for concurrent write-sharing between cooperating clients in the absence of explicit

locking. The fact that write-sharing is infrequent even in those distributed file systems that support it [Welch90] is a reason NFS has been successful despite this limitation. Both Spritely NFS and NQNFS demonstrate how to provide stronger consistency guarantees with a provision for server and client crash recovery. Both approaches depend on the clients to re-establish state after server reboots.

Disconnected operation of fixed and nomadic clients is a potential area for future work. More investigation is required on how consistency guarantees work, if at all, in the presence of clients disconnected longer than the lease terms or callback timeouts used by NQNFS or Spritely NFS, respectively.

Stronger security models in NFS are another area for future work. More research is needed on whether to pursue trusted system support in general.

The problems of consistent name space construction and increased availability are areas of research for future protocol revisions and are perhaps best solved with innovative implementations using existing protocols.

9. Conclusions

The constrained NFS Version 3 effort addressed the following concerns with NFS Version 2:

- 64-bit file sizes are now supported.
- Asynchronous writes increased throughput seven-fold over unaccelerated NFS Version 2 implementations.
- Over-the-wire traffic measured both by RPC counts and network loading has been reduced.
- Directory browsing is faster, with less network loading and lower CPU utilization.
- Performance improvements were achieved despite the size increase of the file attribute structures resulting from 64-bit file size support.
- Many "minor annoyances" of the NFS Version 2 protocol have been corrected.

NFS Version 3 was specified, reviewed, prototyped, verified, and supplied by multiple vendors for external testing in less than 24 months from the initial Boston meetings. At Connectathon in 1993, prototype implementations interoperated successfully. We achieved the goal of providing measurable improvements over NFS Version 2 with little effort required to create an implementation.

There is more work to be done. NFS Version 3 offers the potential for better name and attribute cache management than is possible with NFS Version 2. Realization of this potential is a current and future effort.

9.1. Availability

The NFS Version 3 protocol specification draft can be obtained from `bcm.tmc.edu`, `gatekeeper.dec.com` and `ftp.uu.net` using anonymous FTP.

NFS Version 3 will be available in the next major release of Digital's OSF/1. Servers will fully support NFS Version 3, as well as provide NFS Version 2 for interoperability with older clients. At SunSoft, a Solaris 2 implementation of NFS Version 3 that supports TCP and large transfer sizes is in early deployment and will shortly go to external field test. In addition, a reference implementation of NFS Version 3 with TCP support is undergoing final testing. Early access to the reference implementation from SunSoft will occur this summer. Other implementations are in progress. Contact your vendor for further information.

SunSoft is developing an NFS Version 3 Protocol Validation Suite to provide a tool to help ensure interoperability of clients and servers. This validation suite will be made available for licensing.

10. Acknowledgments

Rusty Sandberg was the author of the earliest NFS Version 3 proposal. Eight specifications intervened between then and now; in many ways we returned to the simplicity of the original. Peter Staubach designed the versions of asynchronous writes, `NFS3ERR_JUKEBOX`, and `READDIRPLUS` described in this paper. Peter also introduced the notion of weak cache consistency in NFS. The Boston group included Cathe Ray, Carl Smith, Peter Staubach, and Brian Pawlowski of SunSoft, Inc., Fred Glover, and Chet Juszczak of Digital, Mark Witte of Data General, John Gillono of Cray Research, Tom Talpey of OSF, and Geoff Arnold of SunSelect, Inc. Spencer Shepler of IBM would have joined us but for his wedding; he did participate in the post-Boston discussion. Chris Duke has been very supportive in his role as the Sun NFS engineering manager. Charlie Briggs at Digital reviewed early drafts and suggested the state diagram. Eric Werme at Digital worked on the server implementation. Michael Kupfer implemented the Network Lock Manager reference source. Brent Callaghan implemented the original MOUNT reference source. Glen Dudek at VGI, our paper shepherd for USENIX, provided invaluable detailed reviews beyond the call of duty. Jeffrey Mogul commented on a very rough early draft. Chris Duke, Byron Rakitzis, Rob Salmon, Dana Treadwell, Rusty Sandberg, Kim Pawlowski, Olga Koudalides, Ellie Koudalides, Michael Eisler, Brian Ehrmantraut, Michael Nelson, Tom Talpey, Bob

Lyon, Cheena Srinivasan, Eric Werme, Michael Kupfer and Tom Tierney reviewed various drafts of this paper on an outrageously compressed schedule. Chad Davies, Richard Binder, and Karla Sorenson greatly improved the readability of this paper. Mike Kazar kept the paper honest.

11. Bibliography

- [Arnold91] Arnold, Geoff, "Change and Non-change in NFS," *Proc. of European Sun Users Group*, 1991. *Discusses changes requiring a protocol revision in NFS.*
- [Bhide91] Bhide, A., Elnozahy, E., Morgan, S., "A Highly Available Network File Server," *Winter USENIX Conference Proceedings*, USENIX Association, Berkeley, CA, January 1991. *Describes an NFS server implementation using redundant servers and dual-ported disks that logs volatile reply cache information to disk.*
- [Fadden92] Fadden, Fran, "Token Mapping Service," Trusted System Interest Group, TSIG document TSIG-TNFS-006.01.01, May 24, 1992. *Description of the security token mapping scheme proposed in TNFS.*
- [Glover92] Glover, Fred, "Request for Comments on a Specification of Trusted NFS (TNFS) Protocol Extensions," Trusted System Interest Group, TSIG document TSIG-TNFS-001.02.02, May 24, 1992. *Proposed draft standard for security extensions to NFS for a trusted environment.*
- [Hitz94] Hitz, D., Lau, J., Malcolm, M., "File System Design for an NFS File Server Appliance," *Winter USENIX Conference Proceedings*, USENIX Association, Berkeley, CA, January 1994. *Describes a highly integrated approach using a log-based file system and nonvolatile RAM to solve the write bottleneck on NFS Version 2.*
- [Howard88] Howard, J.H., M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M.J. West, "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems* 6(1), February, 1988. *Primary reference on the Andrew File System—contrasts the performance and scalability of AFS and NFS—cites the lack of consistency guarantees as the cause of poor scalability of NFS file servers.*
- [Israel89] Israel, Robert K., Sandra Jett, James Pownell, George M. Ericson, "Eliminating Data Copies in UNIX-based NFS Servers," *Uniforum Conference Proceedings*, San Francisco, CA, Feb. 27 - Mar. 2, 1989. *Describes two methods for reducing data copies in NFS server code.*
- [Jacobson88] Jacobson, V., "Congestion Control and Avoidance," *Proc. ACM SIGCOMM '88*, Stanford, CA, August 1988. *Describes TCP performance improvements over WANs and gateways. This work was a starting point for the NFS Dynamic Retransmission work.*
- [Joy84a] Joy, Bill, "Sun Network File Protocol Design Considerations," Internal Sun Microsystems technical note, January 1984. *A description of the basic design principles in the NFS protocol, rationale and implementation, including the use of a reply cache for correctness.*
- [Joy84b] Joy, Bill, "Design of the Sun Network File System," Internal Sun Microsystems technical note, January 1984. *Design of the implementation, relationship to VFS, comparison to ND, omissions in design and reasons, and related work.*

- [Juszczak89] Juszczak, Chet, "Improving the Performance and Correctness of an NFS Server," *USENIX Conference Proceedings*, USENIX Association, Berkeley, CA, January 1989, pages 53-63. *Describes a server reply cache implementation for work avoidance. Listed as a side-effect, the reply cache avoids the destructive re-application of nonidempotent operations—improving correctness.*
- [Juszczak94] Juszczak, Chet, "Improving the Write Performance of an NFS Server" *USENIX Conference Proceedings*, USENIX Association, Berkeley, CA, January 1994. *Describes a write gathering technique that exploits NFS client implementation parallel write behavior to improve write throughput in NFS Version 2.*
- [Kazar88] Kazar, Michael Leon, "Synchronization and Caching Issues in the Andrew File System," *USENIX Conference Proceedings*, USENIX Association, Berkeley, CA, Dallas Winter 1988, pages 27-36. *Describes cache consistency in AFS and contrasts it with other distributed file systems.*
- [Kazar90] Kazar, Michael Leon, Leverett et al., "DEcorum File System Architectural Overview," *USENIX Conference Proceedings*, USENIX Association, Berkeley, CA, Anaheim June 1990. *Describes the DCE DFS file system.*
- [Kazar94] Kazar, Michael, private communication April 1, 1994. *Mike is right—the reply cache is volatile state.*
- [Macklem91] Macklem, Rick, "Lessons Learned Tuning the 4.3BSD Reno Implementation of the NFS Protocol," *Winter USENIX Conference Proceedings*, USENIX Association, Berkeley, CA, January 1991. *Describes performance improvement (reduced CPU loading) through elimination of data copies in tuning the 4.3BSD Reno NFS implementation, and the performance and use of TCP as a transport.*
- [Macklem94] Macklem, Rick, "Not Quite NFS, Soft Cache Consistency for NFS," *Winter USENIX Conference Proceedings*, USENIX Association, Berkeley, CA, January 1994. *Describes a cache consistent NFS protocol, with extensions similar to the work described here.*
- [McVoy91] McVoy, L., Kleiman, S., "Extent-like Performance from a UNIX File System," *Winter USENIX Conference Proceedings*, USENIX Association, Berkeley, CA, January 1991. *Describes a write clustering technique for UNIX local file system writes to improve throughput.*
- [Mogul92] Mogul, Jeffrey C., "A Recovery Protocol for Spritely NFS," *USENIX File System Workshop Proceedings*, Ann Arbor, MI, USENIX Association, Berkeley, CA, May 1992. *Second paper on Spritely NFS proposes a scheme for recovering state in a consistency protocol.*
- [Mogul93] Mogul, Jeffrey C., "Recovery in Spritely NFS," Research Report 93/2, Digital Equipment Corporation Western Research Laboratory, June 1993. *Third paper on Spritely NFS describes the implementation of recovery.*
- [Moran90] Moran, J., Sandberg, R., Coleman, D., Kepecs, J., Lyon, B., "Breaking Through the NFS Performance Barrier," *Proceedings of the 1990 Spring European UNIX Users Group*, Munich, Germany, pages 199-206, April 1990. *Describes the application of nonvolatile RAM in solving the synchronous write bottleneck in NFS Version 2.*
- [NFS3] Sun Microsystems, Inc., "NFS Version 3 Protocol Specification," February 16, 1994.
- [Nelson88a] Nelson, Michael N., Brent B. Welch and John K. Ousterhout, "Caching in the Sprite Network File System," *ACM Transactions on Computer Systems* 6(1). February, 1988. Also *Computing Reviews*, Vol. 30, No. 3, March 1989. *Caching strategies, consistency protocol and performance results.*
- [Nelson88b] Nelson, M.N., "Physical Memory Management in a Network Operating System," *Ph.D. Thesis*. Univ. of Calif., Berkeley. November, 1988.
- [Nowicki89] Nowicki, Bill, "Transport Issues in the Network File System," *ACM SIGCOMM newsletter Computer Communication Review*, April 1989. *A brief description of the basis for the dynamic retransmission work.*
- [Ousterhout90] Ousterhout, John K., "Why aren't Operating Systems Getting Faster as Fast as Hardware," *Proceedings of the 1990 Summer USENIX Conference*, Anaheim, June 11-15, 1990. *A description, in part, of the synchronous write bottleneck in NFS Version 2.*
- [POSIX90] Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language] ISO/IEC 9945-1: 1990, IEEE Std 1003.1-1990.
- [Pawlowski89] Pawlowski, Brian, Ron Hixon, Mark Stein, Joseph Tumminaro, "Network Computing in the UNIX and IBM Mainframe Environment," *Uniforum '89 Conf. Proc.*, (1989). *Description of an NFS server implementation for IBM's MVS operating system.*
- [Presto93] Digital Equipment Corporation. "Guide to Prestoserve," DEC OSF/I Prestoserve Product Documentation, Order number AA-PQTOA-TE, March 1993.
- [RFC1014] Sun Microsystems, Inc., "External Data Representation Specification," RFC-11014, DDN Network Information Center, SRI International, Menlo Park, CA. *Describes canonical data exchange format for use with RPC.*
- [RFC1057] Sun Microsystems, Inc., "Remote Procedure Call Specification," RFC-1057, DDN Network Information Center, SRI International, Menlo Park, CA.
- [RFC1094] Sun Microsystems, Inc., "Network Filesystem Specification," RFC-1094, DDN Network Information Center, SRI International, Menlo Park, CA. *NFS Version 2 protocol specification.*
- [Reid90] Reid, Jim, "N(e)FS: the Protocol is the Problem," *Proc. of the UKUUG Conference*, London, July 1990. *Describes problems in NFS Version 2.*
- [Sandberg85] Sandberg, R., D. Goldberg, S. Kleiman, D. Walsh, B. Lyon, "Design and Implementation of the Sun Network Filesystem," *USENIX Conference Proceedings*, USENIX Association, Berkeley, CA, Summer 1985. *The basic paper describing the SunOS implementation of NFS Version 2.*
- [Satyanarayanan89] Satyanarayanan, M., "A Survey of Distributed File Systems," *Annual Review of Computer Science*, Annual Reviews, Inc. Volume 4, 1989. Also available as Technical Report CMU-CS-89-116, Dept. of Comp. Sci., Carnegie Mellon University. *A survey of NFS, AFS and other distributed file systems with a comprehensive bibliography.*
- [Srinivasan89] Srinivasan, V., Jeffrey C. Mogul, "Spritely NFS: Implementation and Performance of Cache Consistency Protocols", WRL Research Report 89/5,

Digital Equipment Corporation Western Research Laboratory, 100 Hamilton Ave., Palo Alto, CA, 94301, May 1989. Also in Proc. of the Twelfth ACM Symposium on Operating Systems Principles. *Analysis of a Sprite-like consistency protocol applied to NFS.*

[Welch90] Welch, Brent, "Measured Performance of Caching in the Sprite Network File System" *Computing Systems*, Volume 3, Number 4, Summer 1991, pages 315-342. *Analyzes the effectiveness of caching in the Sprite network file system, and the frequency of concurrent write sharing.*

[Wittle93] Wittle, Mark, Bruce Keith, "LADDIS: The Next Generation in NFS File Server Benchmarking" *Proc. Summer 1993 USENIX Conference*, USENIX Association, Cincinnati, OH, June 1993, pages 111-128. *Describes a synthetic, parallel benchmark used to evaluate NFS Version 2 server performance.*

[X/OpenNFS] X/Open Company, Ltd., X/Open CAE Specification: Protocols for X/Open Internetworking: XNFS, X/Open Company, Ltd., Apex Plaza, Forbury Road, Reading Berkshire, RG1 1AX, United Kingdom, 1991. *Describes the NFS version 2 protocol and accompanying protocols, including the Lock Manager and the Portmapper.*

[X/OpenPCNFS] X/Open Company, Ltd., X/Open CAE Specification: Protocols for X/Open Internetworking: (PC)NFS, Developer's Specification, X/Open Company, Ltd., Apex Plaza, Forbury Road, Reading Berkshire, RG1 1AX, United Kingdom, 1991.

Author information

Chet Juszczak is a Consultant Engineer in the UNIX Software Group at Digital where he works on distributed file systems for commercial servers. He has worked on NFS and file server performance at Digital since 1985. Before that he worked on relational database technologies at AT&T. Chet got his M.S. in C.S. at the University of Michigan in 1983. Reach him electronically at chet@zk3.dec.com or via U.S. Mail at Digital Equipment Corp., 110 Spit Brook Rd., Nashua, NH 03062.

Brian Pawlowski now resides at Network Appliance Corp., where he works on performance and new architectures for appliances. Brian did the work described here while at Sun Microsystems, Inc., where for six years he worked on distributed file systems. He led the MVS/NFS project, slummed with AFS and DCE DFS, worked on LADDIS and finally became aware while watching the pretty colors during a multiprocessor file server performance project that provided a convenient excuse to use some sophisticated performance visualization tools. Brian has never castrated nor slaughtered cattle. Reach him electronically at beepy@netapp.com or via snail mail at Network Appliance, 295 N. Bernardo Ave., Mountain View, CA 94043.

Peter Staubach is a staff engineer at SunSoft Inc. He is the project leader, principal designer and engineer on the NFS Version 3 and NFS over TCP projects. Peter has been involved with the design and implementation of SunSoft's distributed file system since 1991. Prior to joining Sun, Peter worked at Lachman Associates where he was involved with the implementation of NFS for System V and sundry NFS ports. He can be reached electronically at staubach@eng.sun.com or via U.S. Mail at SunSoft Inc. 2550 Garcia Ave. MS MTV05-40, Mountain View, CA

94043.

Carl Smith has been a member of the technical staff at Sun for six years. During this time he has been the NFS project lead, NFS Version 3 co-conspirator, and worked on various aspects of NFS, RPC, and TCP/IP. His friends have observed that he is working his way down the protocol stack. Prior to Sun he was a jack-of-all-trades at UniSoft Corporation and the technical manager and a principal engineer of the Berkeley PDP-11 Software Distribution. He has been involved with UNIX since Version 6. His current interests are networking, security, and retiring early to visit the bookstores of the world. He can be reached electronically at cs@eng.sun.com or via U.S. Mail at SunSoft Inc. 2550 Garcia Ave. MS MTV05-44, Mountain View, CA 94043

Diane Lebel is an engineer in the UNIX Software Group at Digital where she is the principal designer of the DEC OSF/I NFS client. Diane has worked at Digital since 1987. Reach her electronically at lebel@zk3.dec.com or via U.S. Mail at Digital Equipment Corp., 110 Spit Brook Rd., Nashua, NH 03062.

David Hitz is a co-founder and director of system architecture at Network Appliance Corp. Dave has focused on designing and implementing the WAFL file system, and on the overall design of their file server. He also worked at Auspex Systems in the file system group, and at MIPS in the System V kernel group. He received his computer science BSE from Princeton University in 1986. Reach him electronically at hitz@netapp.com or via snail mail at Network Appliance, 295 N. Bernardo Ave., Mountain View, CA 94043.

Trademarks

NFS is a trademark of Sun Microsystems, Inc. UNIX is a registered trademark of UNIX System Laboratories, a wholly-owned subsidiary of Novell, Inc. Prestoserve is a trademark of Legato Systems, Inc.

Appendix 1. Summary of the NFS Version 3 protocol. Notes: `fh` = file handle, `dir_fh` = directory file handle, `wcc_data` = weak cache consistency data.

Procedure	Arguments	Returns	Change from NFS Version 2
<code>ONULL—do no work</code>	<code>none</code>	<code>none</code>	
<code>1 GETATTR—Get attributes of a file</code>		attributes	File types are defined for sockets and named pipes as well as block and character devices, with explicit subfields for major and minor device numbers.
<code>2 SETATTR—Set attributes of a file</code>	<code>fh, new_attributes</code>	<code>wcc_data</code>	Now uses a discriminated union for each field, telling whether the field is to be set (and how), eliminating unsigned field overloading with -1. The time fields can be set to either the server's current time or the time supplied by the client.
<code>3 LOOKUP a file name</code>	<code>dir_fh, name</code>	<code>fh, attributes, directory_attributes</code>	Returns the attributes for the directory searched for more efficient name cache validation.
<code>4 ACCESS—check access permissions</code>	<code>fh, access mask</code>	<code>attributes, access mask</code>	New procedure, allows over-the-wire access check at open to detect permission failures. Also useful for ACL-based implementations.
<code>5 READLINK—read from a symbolic link</code>	<code>fh</code>	<code>attributes, symlink data</code>	Returns attributes for symbolic link.
<code>6 READ data from a file</code>	<code>fh, offset, count</code>	<code>attributes, count, eof_flag, data</code>	Reply structure includes an EOF boolean allowing clients to determine efficiently the end of file.
<code>7 WRITE data to a file</code>	<code>fh, offset, count, storable flag, data</code>	<code>count, committed_flag, verifier, wcc_data</code>	An enumerated field indicates disposition of request. Removed the begin_offset and total_count fields in the NFS Version 2 protocol. The reply contains a count so that the server can write less than the requested amount and correctly indicate this to the client.
<code>8 CREATE a file</code>	<code>dir_fh, name, attributes to apply</code>	<code>fh, attributes, wcc_data</code>	Contains an exclusive flag to support exclusive creation of regular files (in conjunction with an exclusive create verifier). Returns <code>wcc_data</code> for directory.
<code>9 MKDIR—create a directory</code>	<code>dir_fh, name, attributes to apply</code>	<code>fh, attributes, wcc_data</code>	Returns <code>wcc_data</code> for parent directory.
<code>10 SYMLINK—Create a symbolic link</code>	<code>dir_fh, name, attributes to apply, data</code>	<code>fh, attributes, wcc_data</code>	Returns <code>wcc_data</code> for directory and a file handle and attributes for the symlink.
<code>11 MKNOD—Create a special device</code>	<code>dir_fh, name, attributes to apply, special data</code>	<code>fh, attributes, wcc_data</code>	New procedure allows creation of special files without overloading the fields in the CREATE procedure.
<code>12 REMOVE a file</code>	<code>dir_fh, name</code>	<code>wcc_data</code>	Returns <code>wcc_data</code> for directory.
<code>13 RMDIR—remove a directory</code>	<code>dir_fh, name</code>	<code>wcc_data</code>	Returns <code>wcc_data</code> for parent directory.
<code>14 RENAME a file</code>	<code>source dir_fh, source name, target dir_fh, target name</code>	<code>wcc_data, wcc_data</code>	Returns <code>wcc_data</code> for both directories.
<code>15 LINK—create a hard link</code>	<code>fh, target dir_fh, name</code>	<code>attributes, wcc_data</code>	Returns <code>wcc_data</code> for directory.
<code>16 READDIR—read directory entries</code>	<code>fh, cookie, verifier, count</code>	<code>directory_attributes, verifier, entries</code>	Arguments include a verifier to allow the server to validate the directory offset. Returns attributes for directory
<code>17 READDIRPLUS—Read directory entries/attributes</code>	<code>fh, cookie, verifier, count, maximum size</code>	<code>directory_attributes, verifier, entries</code>	New procedure returns file handle and attributes in addition to directory entries.
<code>18 FSSSTAT—volatile file system information</code>	<code>fh</code>	<code>attributes, file system utilization</code>	Reply includes total size and free space in a file system in bytes (removing the ambiguous notion of blocks as the file system size), total number of files and free file slots, and an estimate of the time between file system modifications (for cache consistency checking algorithms).
<code>19 FSTINFO—Get static file system information</code>	<code>fh</code>	<code>attributes, file system properties</code>	Provides nonvolatile file system information including preferred/maximum read/write transfer size, and flags stating whether links or symbolic links are supported. Also includes preferred transfer sizes for READDIR and READDIRPLUS, the server time granularity, and an indication of whether times may be set in a SETATTR request.
<code>20 PATHCONF</code>	<code>fh</code>	<code>attributes, POSIX information</code>	Retrieve POSIX info—maximum name length, number links, etc. supported by remote file system.
<code>21 COMMIT data on server to stable storage</code>	<code>fh, offset, count</code>	<code>wcc_data, verifier</code>	This procedure has been removed from the MOUNT protocol and put in NFS where it belongs. Provides the synchronization mechanism for asynchronous writes.

Clue Tables: A Distributed, Dynamic-Binding Naming Mechanism*

Cheng-Zen Yang, Chih-Chung Chen, and Yen-Jen Oyang

Department of Computer Science
and Information Engineering
National Taiwan University
Taipei, Taiwan, R.O.C.

Abstract

This paper presents a distributed, dynamic naming mechanism called clue tables for building highly scalable, highly available distributed file systems. The clue tables naming mechanism is distinctive in three aspects. First, it is designed to cope well with the hierarchical structure of the modern large-scale computer networks. Second, it implicitly carries out load balancing among servers to improve system scalability. Third, it supports file replication and dynamically designates a primary copy to resolve possible data inconsistency. This paper also reports a performance evaluation of the clue tables mechanism when compared with NFS, a popular distributed file system.

1 Introduction

Distributed file systems are the backbone of the modern network computing environment. The naming mechanism in a distributed file system maps the logical name of each individual file to its physical location. In the design of a modern distributed file system, availability and scalability are two essential concerns [1, 2]. In order to build a highly available, highly scalable distributed file system, a designer must incorporate a naming mechanism that can cope with these concerns.

To meet the demands, a naming mechanism must be distributed in nature, and support file replication and dynamic binding. The naming mechanism must be distributed in nature because centralized naming mechanisms suffer low scalability due to limited capacity of the central naming server. The naming mechanism must support file replication because file replication improves both availability and scalability of the system. The presence of replicated file copies prevents

service disruption due to failure of a single file server and thus improves system availability. With file replication, the scalability of the system is improved because clients can access replicated file copies on different servers to avoid congestion of a particular file server. To maximize the benefits of file replication, the naming mechanism must support dynamic binding. With dynamic binding, the clients that initially turn to a crashed server for file service can establish new connections on-the-fly to other servers that have replicated copies of the files. Also, clients can dynamically select a server for binding to achieve a good load balancing among servers.

In this paper, we propose a new distributed, dynamic-binding naming mechanism called clue tables. The clue tables mechanism offers the basis to build a highly available, highly scalable distributed file system and is distinctive in three aspects:

1. It is designed to cope with the hierarchical structure of modern computer networks –
In a modern computer network, particularly a large-scale computer network, bridges are commonly installed to partition the network into a number of clusters. For example, the network in a research institute may be partitioned so that the computers in each laboratory form a local cluster. The hierarchy of network partitions may extend over several levels. The major distinction of the clue tables mechanism is that it was designed to cope with the hierarchical network structure. With clue tables, we can make clients turn first to local file servers to locate a file. If a client can not find the file on local servers, or the local servers that store the file are unavailable, e.g. crashed, then the client will automatically go one level up in the network hierarchy to locate the file on remote servers. The main reason behind

*This research was sponsored by National Science Council of R.O.C. under grant NSC 83-0408-E-002-002

adopting this practice is to make most bindings among clients and servers occur in conformity with network hierarchy design.

In reality, this is the main distinction between the clue tables mechanism and the similar prefix tables mechanism [3, 4]. With prefix tables, a client that is bound to a remote server for file service due to failure of local servers will not automatically switch back to local servers upon successful recovery of local servers. As a result, a client may still rely heavily on a remote server for file service for a long period of time after a crashed local server is back to work again. On the other hand, with clue tables, a client will always turn to local servers first to locate a file when it starts a new file session, i.e. opens a file. This guarantees that the bindings among clients and servers occur in conformity with network hierarchy design whenever possible.

2. It implicitly carries out load balancing among servers to improve system scalability –

The clue tables mechanism implements dynamic binding between clients and servers upon file open to achieve load balancing among servers. This is the main distinction of the clue tables mechanism when compared with other distributed file systems such as AFS[1, 5], Coda[6], Locus[7], V kernel[8], Amoeba[9], Ficus[10], and Deceit[11] that also implement dynamic binding. With clue tables, a client, upon opening a file, multicasts access requests to servers that have a replica of the file. If more than one server acknowledges the request, the client always chooses the server that responds fastest for binding. With this practice, the clue tables mechanism implicitly carries out load balancing among servers since a server with a lighter load has a better chance to respond faster than a server with a heavier load. Though it is not guaranteed that the server with the lightest load among multiple candidates always responds fastest, a near-optimum load balancing situation should persist most of time.

3. It supports file replication and dynamically designates a primary file copy to resolve possible data inconsistency –

One crucial issue with file replication is how to maintain data consistency. The clue tables mechanism resolves this issue by dynamically designating a primary file copy. It is the dynamic nature and granularity of the mechanism that distinguishes the clue tables mechanism from other distributed file systems that also employ a pri-

mary copy based approach, e.g. Locus[7]. The main reason behind employing the dynamic approach with file-level granularity is to distribute servers' load. With clue tables, when one or more clients attempt to write to a file, one server is dynamically designated as the primary server and all accesses to the file are temporarily forwarded to the primary server. Once the situation that could cause data inconsistency no longer exists, the primary server broadcasts the new version of the file to other servers.

In the following part of the paper, section 2 describes the basic structure of clue tables. Section 3 elaborates on the system operations with clue tables. Section 4 addresses the implementation and performance issues. Section 5 concludes this paper.

2 Basic Structure of Clue Tables

The clue tables mechanism implements a global naming space. The primitive entities in the clue tables mechanism are file collections termed *domains*. A domain is a subtree in the integrated file system of a distributed system. A file server can contain one or more domains while a domain cannot spread over multiple file servers. A domain must be entirely stored on one file server. However, we may have replicated copies of a domain stored in a number of servers. Fig. 1 illustrates the naming architecture with clue tables.

Each file in a domain is an object composed of two attributes: logical file name and physical location. Each file is uniquely identified by its logical file name in the integrated file system. The physical location attribute specifies where the file resides. Clue tables are the directories that the clients refer to for locating a file based on its logical file name. Fig. 2 shows an example of clue tables. A clue table contains a number of entries, each of which corresponds to a domain in the integrated distributed file system.

For example, in Fig. 2, there are two entries. The first entry corresponds to the domain with root "/usr" and the second entry is corresponds to the domain with root "/usr/bin". A clue table entry specifies the file servers that have a copy of the domain. For example, the domain with root "/usr" has three replicated copies on file servers **solar**, **earth**, and **global**.

With file replication, the availability and scalability of the system is significantly enhanced. The multiple servers listed in a clue table entry are grouped and prioritized. For example, in the first entry of Fig. 2, servers **solar** and **earth** form the first group, which is separated from the second group by a semicolon. The

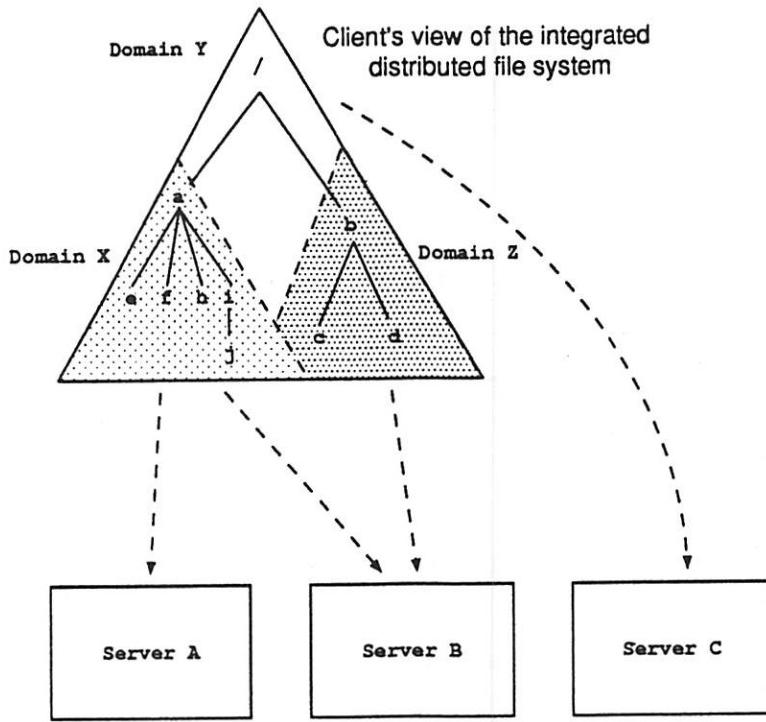


Figure 1: The naming architecture. Domain X is stored on servers A and B. Domain Y is stored on server C. Domain Z is stored on server B.

second group contains only one server, **global**. When trying to locate a file, a client first turns to the servers in the first group. If all the servers in this group are unavailable, e.g. crashed or unreachable due to network failure, the client then turns to the second group and so on. The motivation to group and prioritize servers in the list is to make the bindings among clients and servers occur in conformity with network hierarchy design whenever possible.

A clue table entry is overridden by another entry when the second entry is with root a subdirectory of the first domain. For example, in Fig. 2, the second entry, corresponding to the domain rooted by “/usr/bin”, overrides the first entry. When a client tries to locate a file, it first searches the clue table for the longest prefix of the domain that matches the filename. The client then sends requests to the servers in the list.

A clue aliasing mechanism is incorporated to provide more flexibility in system integration. The second entry in Fig. 2 shows an example of clue aliasing. When the client accesses directory “/usr/bin” and sends a request to server csm (see Fig. 3), it is actually accessing directory “/usr/sparc/bin” on csm. Through clue aliasing, a directory on a server can appear to have different names on different clients. This adds desirable flexi-

bility to system integration.

Certain rules apply to the creation of clue tables in a distributed file system:

1. Each node, a client or server, in a distributed system should have a clue table. A clue table can be shared by two or more nodes but each node must have access to one clue table. The clue table of a node can be stored in a node’s local disk if it has one. If the node is diskless, then the clue table is stored in a remote server and is cached by the node in the memory while it is operating.
2. Each node may have some private entries in its own clue table. One good use of this flexibility is the creation of the private “/tmp” directory. If a client has local disks, it may be more appropriate, from the performance aspect, to place temporary files created by this client on its local disks rather than on remote servers. Even if the client does not have a local disk, it may share a localized “/tmp” directory with other diskless clients in the local cluster.
3. If a domain is shared by multiple nodes, all the nodes must contain the same set of servers in their

```

/usr :rep=3:      # three replicated copies.
    solar,      # server name list.
    earth;
    global.     # second group.
/usr/bin :rep=3:  # three replicated copies.
    earth,
    solar,
    csm=/usr/sparc/bin.# clue aliasing.
    :

```

Figure 2: An example of clue tables and clue aliasing.

clue table entries corresponding to this domain. The grouping and prioritizing of these servers may be different, reflecting each node's physical position in the network hierarchy. However, the set of servers in the entries must be identical. Otherwise, data consistency among replicas of the domain cannot be maintained. (Detailed discussion on data consistency guarantees is presented in next section.)

4. When creating a clue table for a node, we should group and prioritize the servers according to their proximity to the node in the network hierarchy. By doing so, the node will always find a file in the nearest available server and its interference with remote nodes in the network will be minimized.

3 System Operations with Clue Tables

This section discusses how the system operates with clue tables. Since the clue tables mechanism implements dynamic binding, the bindings among clients and servers can change on-the-fly. A client establishes a binding to a server upon the start of a file session. A file session is a series of file operations to a file enclosed by open and close operations. When a client starts a file session, it first searches the clue table for the domain that includes the file and sends requests to the servers according to the grouping and priority specified in the clue table. The servers that receive the request respond by locating the file in their own storage and returning a *succeeded* or *failed* message. If replicated copies exist on several servers, the client will choose the server that responds most quickly to a succeeded message for service of this file session. Fig. 3 illustrates the access request flow.

As mentioned earlier, through implementing dynamic binding upon file opening, the clue tables mech-

anism implicitly carries out load balancing among servers. It is conceivable that a server with a lighter load has a better chance to respond faster than a server with a heavier load. Though it is not guaranteed that the server with the lightest load among multiple candidates always responds fastest, a near-optimum load balancing situation should persist most of time.

When a client selects a particular server for a file session, the client caches the attribute block of the file, the server identification, and a file handle returned by the server to speed up following file operations. The file handle is a unique file index assigned by the server for speedily identifying and locating an opened file. Note that the bindings among clients and servers are per file session basis. A client may turn to different servers for service of different file sessions. Due to network proximity and server load, two clients that are concurrently accessing the same file may be served by two different servers. As a result, access load is distributed over servers and the scalability of the system is significantly upgraded.

The bindings among clients and servers may change during a file session. One reason is to elide access disruption caused by server or network failure. When such a failure occurs, the client will search the clue table for another server that has a replicated copy of the file. If this search succeeds, the client will establish a binding with the second server and the user will not observe service disruption except that the latency of some file operations is longer.

Another occasion in which rebinding is invoked is to maintain data consistency among replicated file copies. With file replication, the system must be able to resolve potential data inconsistency when concurrent writing or read-write sharing occurs.

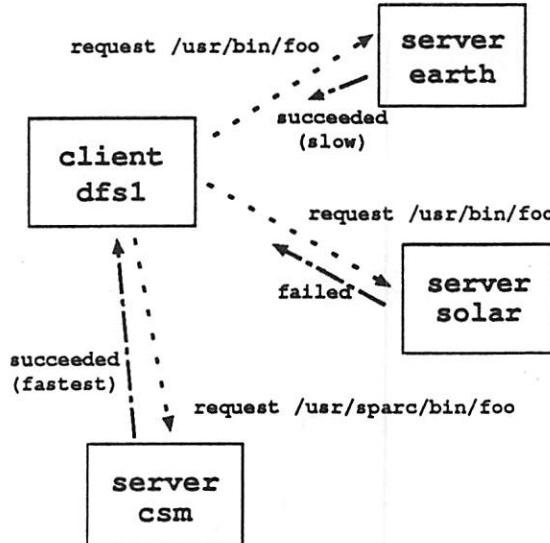


Figure 3: Flow of access requests. In this case, server **csm** is the fastest to reply a *succeeded* message. Client **dfs1** will establish a binding to **csm**'s */usr/sparc/bin/foo*.

We addressed the data consistency issue by introducing a primary copy based preventive mechanism. When one or more clients open a file for writing, a server that holds a replica of the file is designated as the primary server. All other servers that also have a replica of the file invalidate their copies. Meanwhile, those clients that are initially bound with the servers that have temporarily invalidated copies will carry out rebinding operations to connect to the primary server. The primary server will provide all accessing services to the file as long as the writing operations continue. Upon termination of the situation, the primary server will broadcast the new version of the file to other servers.

An interesting issue here is how the primary server is selected. As mentioned earlier, the clue tables mechanism dynamically designates the primary server to distribute servers' load. In the situation that only one client attempts to write to the file, the server that receives the write request will become the primary server. If two or more clients want to open the file for writing at the same time, all the servers that receive a write request will compete to become the primary server. A simple arbitration mechanism based on a pre-assigned priority is used to determine the primary server.

4 Implementation and Performance Evaluation

Fig. 4 shows the structure of an implementation of the clue tables mechanism. This implementation is

based on Mach 2.6 operating system [12] and the VFS (Virtual File System) [13, 14]. On the client side, the VFS(Virtual File System) forwards file accesses that invoke the clue tables mechanism to the CluFS interface. The CluFS interface checks whether the file access hits the local disk/file cache. If not, the file access request is forwarded to a user-level process called the client daemon. The client daemon looks up the local clue table and multicasts the request to the servers according to the grouping and priority in the matched clue table entry. On the server side, the incoming requests are processed by a user-level process called the server daemon. The server daemon interfaces with the VFS to locate the file and returns a *succeeded* or *failed* message to the requesting client.

To study the performance with the clue tables mechanism, we have conducted an experiment and compared the results with NFS[15]. The experimental system consists of five Intel 80486 based personal computers connected by an ethernet network. All machines run Mach 2.6 operating system and three out of the five act as servers while the remaining two act as clients.

Fig. 5 shows the results from the experiment. In the experiment, we repeatedly open and close 30 files to test the overhead of dynamic binding operations. The horizontal axis gives the number of times the operations are repeated. The vertical axis is time the operations take in seconds. Fig. 5 shows that the clue tables mechanism performs slightly better than NFS in file opens.

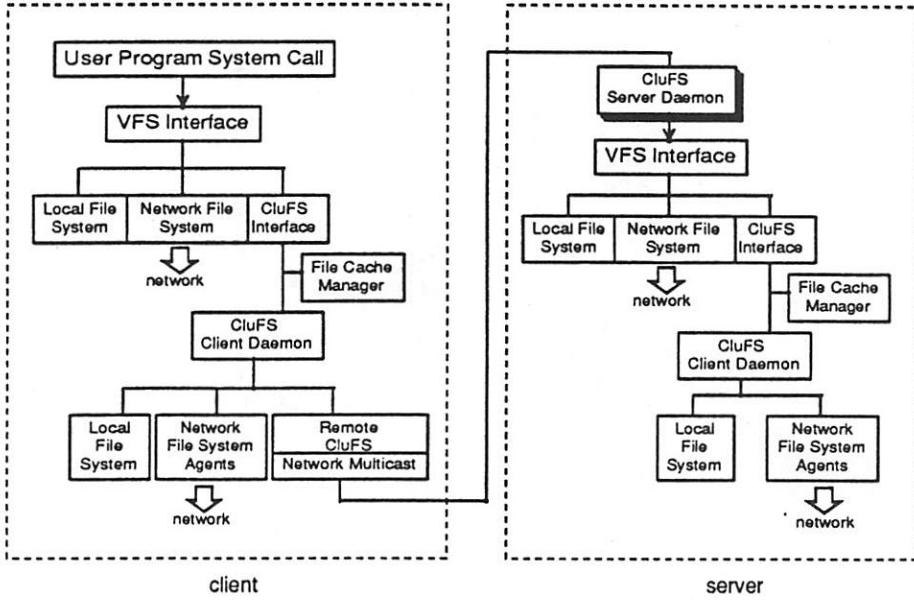


Figure 4: An implementation of the clue tables mechanism.

This is due to the use of the socket mechanism[16] in the clue tables file system. The socket mechanism induces less overhead than the RPC (Remote Procedure Call) mechanism used in NFS. For file closes, NFS virtually takes no time. The reason is that NFS uses a stateless file cache coherence protocol and, as a result, does not invoke a remote procedure call when a file is closed.

5 Conclusion

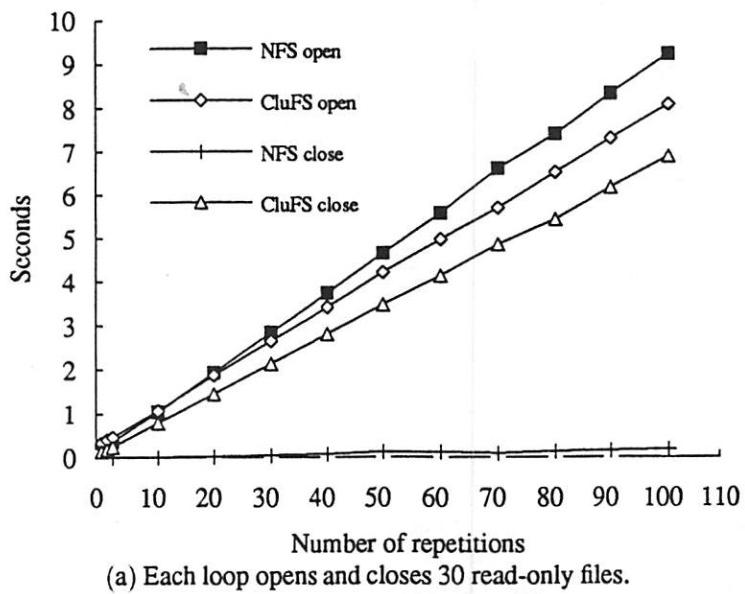
In this paper, we presented a distributed, dynamic naming mechanism called clue tables for building highly scalable, highly available distributed file systems. The clue tables naming mechanism is distinctive in three aspects. First, it is designed to cope well with the hierarchical structure of modern large-scale computer networks. Second, it implicitly carries out load balancing among servers. Third, it supports file replication and dynamically designates a primary copy to resolve possible data inconsistency caused by concurrent accesses to multiple file replicas. The clue tables naming mechanism is incorporated in the Azalea distributed file system currently being developed at National Taiwan University.

Acknowledgments

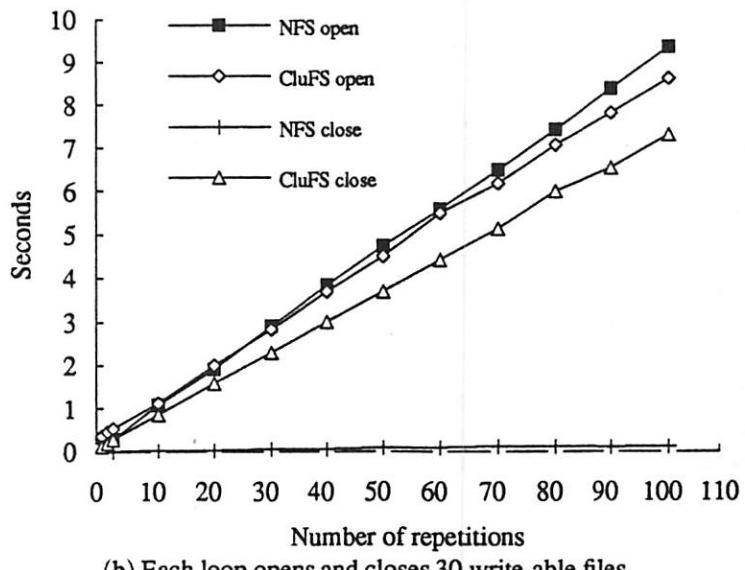
We wish to thank Mary Baker, the shepherd of this paper, and the reviewers for many helpful suggestions and comments.

References

- [1] Mahadev Satyanarayanan. Scalable, Secure, and Highly Available Distributed File Access. *IEEE Computer*, 23(5):9–21, May 1990.
- [2] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John Ousterhout. Measurements of a Distributed File System. In *Proc. of the 13th Symposium on Operating Systems Principles*, pages 198–212. ACM, 1991.
- [3] Brent Welch and John Ousterhout. Prefix Tables: A Simple Mechanism for Locating Files in a Distributed System. In *The 6th Int'l. Conference on Distributed Computing Systems*, pages 184–189. IEEE, May 1986.
- [4] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite Network Operating System. *IEEE Computer*, 21(2):23–36, Feb. 1988.
- [5] Mahadev Satyanarayanan, John H. Howard, David A Nichols, Robert N. Sidebotham, Alfred Z. Spector, and Michael J. West. The ITC Distributed File System: Principles and Design. In *Proc. of the 10th Symposium on Operating Systems Principles*, pages 35–50. ACM, Dec. 1985.
- [6] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A Highly Available File System for a Distributed Workstation Envi-



(a) Each loop opens and closes 30 read-only files.



(b) Each loop opens and closes 30 writeable files.

Figure 5: 30 different files are repetitively opened and closed.

- ronment. *IEEE Tans. on Computer*, 39(4):447–459, Apr. 1990.
- [7] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS Distributed Operating System. In *Proc. of the 9th Symposium on Operating Systems Principles*, pages 49–70. ACM, Oct. 1983.
 - [8] David R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314–333, March 1988.
 - [9] S. J. Mullender and A. S. Tanenbaum. A Distributed File Service Based on Optimistic Concurrency Control. In *Proc. of the 10th Symposium on Operating Systems Principles*, pages 51–62. ACM, Dec. 1985.
 - [10] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Gerald J. Popek, and Dieter Rothmeier. Implementation of the Ficus Replicated File System. In *USENIX Summer '90*, pages 63–71, 1990.
 - [11] Alex Siegel, Kenneth Birman, and Keith Marzullo. Deceit: A Flexible Distributed File System. In *USENIX Summer '90*, pages 51–61, 1990.
 - [12] J. Boykin and A. Langerman. Mach/4.3BSD: A Conservative Approach to Parallelization. *Computer Systems*, 3:69–99, 1990.
 - [13] Steve R. Kleiman. Vnode: An Architecture for Multiple File System Types in SUN UNIX. In *Summer USENIX Conference proceedings*, pages 238–247, Atlanta, GA, June 1986.
 - [14] David S. H. Rosenthal. Evolving the Vnode Interface. In *Summer USENIX Conference proceedings*, pages 107–118, Anaheim, CA, 1990.
 - [15] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network File System. In *USENIX Conference proceedings*, pages 119–130, June 1985.
 - [16] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quaterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley Publishing Company, 1989.

Author Information

Cheng-Zen Yang is currently a Ph.D. student at the Dept. of Computer Science and Information Engineering of National Taiwan University, Taipei, Taiwan. He received the B.S. degree in Computer Engineering from National Chiao Tung University in 1988, and

M.S. degree in Computer Science and Information Engineering from the same university in 1990. His major research interests include distributed computing systems and operating systems. He can be reached by e-mail at dennis@solar.csie.ntu.edu.tw.

Chih-Chung Cheng was born in Tainan, Taiwan, R.O.C. He received the B.S. degree in Computer Science and Information Engineering from National Taiwan University in 1992. He is currently in the master program of the same department and expected to complete his degree in 1994. His research interests are distributed file systems and computer networks. He can be reached by e-mail at chung@solar.csie.ntu.edu.tw.

Yen-Jen Oyang received the B.S. degree in Information Engineering from National Taiwan University in 1982, the M.S. degree in Computer Science from the California Institute of Technology in 1984, and the Ph.D. degree in Electrical Engineering from Stanford University in 1988. He is currently an Associate Professor in the Department of Computer Science and Information Engineering, National Taiwan University. His research interests include computer architecture, distributed systems, and VLSI system design. He can be reached by e-mail at yjoyang@csie.ntu.edu.tw.

Optimistic Lookup of Whole NFS Paths in a Single Operation

Dan Duchamp
Columbia University

Abstract

VFS lookup code examines and translates path names one component at a time, checking for special cases such as mount points and symlinks. VFS calls the NFS lookup operation as necessary. NFS employs caching to reduce the number of lookup operations that go to the server. However, when part or all of a path is not cached, NFS lookup operations go back to the server. Although NFS's caching is effective, component-by-component translation of an uncached path is inefficient, enough so that lookup is typically the operation most commonly processed by servers. We study the effect of augmenting the VFS lookup algorithm and the NFS protocol so that a client can ask a server to translate an entire path in a single operation. The preconditions for a successful request are usually but not always satisfied, so the algorithm is optimistic. This small change can deliver substantial improvements in client latency and server load.

1 Introduction

The NFS `lookup` operation frequently goes “over the wire” from client to server. For example, on the main file servers of Columbia’s Computer Science department, lookups constitute approximately 31% of all NFS operations serviced. This makes `lookup` the most common operation in our environment, followed closely by `null` and `getattr`, and then by `read`.¹ Similar results are typical at other installations; `lookup` is the most

¹This data was gathered by the `nfssstat` utility on eight file servers, all running SunOS version 4.1.3 and NFS version 2. The total number of NFS operations, including `null`, was nearly 4 million. The frequency of the other common operations was 27.5%, 22.7%, and 4.8% for `null`, `getattr`, and `read`, respectively. Across the eight servers there was considerable variance among the relative frequencies, but, in every case, `lookup`, `getattr`, and `null` were by far the most common operations.

common, or at least one of the two or three most common, NFS operation to reach the server.²

These results are obtained despite the presence of a cache (called “directory-name lookup cache,” or DNLC) that is quite effective in mapping a (*directory-vnode*, *name-within-directory*) tuple to the vnode for the name. Measuring in the same environment, we found an average DNLC hit rate of 75% for name lookups in NFS file systems. Apparently, the NFS client side calls `lookup` so often that a quarter of the calls (namely, the DNLC misses) are sufficient, by themselves, to make `lookup` the most common operation at the server.

The seemingly high number of over-the-wire lookups has led us to wonder if they are all necessary and whether some steps might be taken to reduce their number. The most obvious approach is to increase the effectiveness of DNLC. We did this, in two ways:

1. The DNLC implementation of SunOS 4.1.3 will not cache a (*directory-vnode*, *name-within-directory*) tuple if the name is more than 15 characters long. Preliminary measurements indicated that a non-negligible fraction (12%) of DNLC misses were caused by component³ names being longer than 15 characters. Accordingly, we increased the maximum name size to 31 characters.

This change reduced to zero the number of DNLC misses due to over-long component names. However, the effect on the number of NFS lookups was negligible (a fraction of a percent). Investigation revealed

²For example, the most common operations in the `nhfs-stone` benchmark are, in order: `lookup` (34%), `read` (22%), `write` (15%), and `getattr` (13%).

³Following convention, we call the argument to VFS `lookup` a *path*, or *pathname*. A path consists of a sequence of *components*. The process of mapping a path or a component to a vnode we call *translation* or *resolution*.

that over-long names occurred predominantly in the local UNIX file system (UFS) rather than in remote NFS file systems.⁴ This finding is obviously site-dependent and workload-dependent, so it might still be worthwhile to raise the 15-character limit, though perhaps to some number smaller than 31.

2. In the typical configuration of SunOS 4.1.3, the size of DNLC is set according to the formula “17*MAXUSERS + 90.” MAXUSERS is set to 48, leading to 906 cache entries. We doubled this number, with the result of increasing the hit rate for NFS lookups by about half a percent.

The two changes together resulted in increasing the DNLC hit rate for NFS lookups by less than one percent. We conclude that most lookup operations that go to the server are for pathnames that have not been looked up before, or else were looked up in the “distant past.”

So the simple approach of increasing DNLC size will, by itself, not substantially reduce lookup traffic to the server. This should not be surprising, since DNLC has been available for many years, and its performance has presumably been tuned with some care. At least in our environment, it seems that the size of DNLC has been set to beyond the point of diminishing returns.

To substantially reduce lookup traffic to the server requires a more efficient method for looking up “new” pathnames. In the next two sections we describe and evaluate such a method.

2 Path Lookup Algorithm

Roughly speaking, the existing lookup algorithm used at the VFS level is:

```
dir = vnode for start of path;
for (;;) {
    component = next_component(path);
    if (component is ..) {
        if (goes beyond process' root)
            return error;
        while (dir is a mount point) {
            dir = cross back over mount point;
            if (goes beyond process' root)
                return error;
        }
    }
    vnode = VOP_LOOKUP(dir, component);
    if (reached end of path)
        return vnode;
```

⁴The DNLC module is defined at the VFS level, and is callable by any underlying file system, such as NFS or UFS.

```
while (vnode is mounted on)
    vnode = root of overlaid f/s;
if (vnode is symlink)
    prepend symlink to remaining path;
else
    dir = vnode;
}
```

The VOP_LOOKUP macro expands to call the lookup operation of the right type of underlying file system (e.g., NFS, HSFS,⁵ etc.). That operation may use DNLC to reduce the number of lookup calls that go to the server; for example, both NFS and UFS do this.

This algorithm translates component names into vnodes one-by-one, testing for three major special cases at each iteration:

1. the vnode is a symlink
2. the vnode is mounted-on
3. the component is “..”

These special cases form the main reason why lookup happens one component at a time. Symlinks are hardest to handle, since they are a source of uncertainty. That is, a component cannot be known to be a symlink until the server indicates that it is, and expansion of the symlink can change the path arbitrarily. In particular, the unpredictability of the content of symlinks means that not all mount points are evident in a pathname when lookup begins. Crossing a mount point is a major operation, as it potentially changes the server to which lookup operations should be directed. Finally, references to the parent directory (i.e., “..” or “dot-dot”) might also lead to crossing a mount point (in the “up” direction, as opposed to the “down” direction of the previous case).

An additional reason why the VFS lookup algorithm proceeds component-by-component is that the NFS protocol has been designed not to contain pathname syntax in the protocol because of the desirability of keeping operating system dependent detail out of the protocol specification [7]. Since NFS and UFS are the major file systems below the VFS layer, VFS algorithms have been designed to cater to their constraints.

2.1 Overview

The design of the VFS lookup algorithm is sensible, since every component must be checked for the special cases. However, the component-by-component analysis of the path is the cause of the large number of lookups that go to the server. If there were no special cases, then whole paths could be looked up in a single server operation.

⁵The High Sierra file system, for CD-ROM.

In fact, the special cases seldom arise. Measuring in the same environment mentioned earlier — name lookups generated by a multi-user workload applied to eight servers over several days — we found that 97.7% of paths resolved by the VFS lookup algorithm crossed no mount points and 98.9% contained no symlinks.

Our work capitalizes on these facts. We develop a “path lookup” operation that can translate several components of a path. This operation assumes that the path includes no special cases. After a `path-lookup`, we apply some checks for the special cases. If any is found, then further `path-lookups` may be necessary, and it is possible that some of the work performed by the first `path-lookup` may have been wasted. Hence, `path-lookup` is an “optimistic” operation. The number of `path-lookup` operations and the extent to which some of them may perform wasted work varies for each path. However, for the overwhelming majority of paths, a single `path-lookup` suffices to translate the path into a vnode. At worst, the `path-lookup` call will translate only the first component; so the ordinary `lookup` operation is the degenerate case of `path-lookup`.

Our approach is, first, to add a path-to-vnode cache at the VFS level and, second, to augment NFS as necessary to lookup whole paths whenever the path cache misses. Specifically, an additional `path-lookup` call is added to the NFS protocol; this call accepts a pathname which the server translates until the first symlink (if any) is encountered. The response contains three fields:

1. The longest symlink-free prefix of the path. The prefix may be null.
2. The file handle for the prefix.
3. The untranslated suffix of the path, with the first symlink expanded and prepended. The suffix will be null if the path contains no symlink.

Note that our additional VFS-level path cache is separate from and logically above DNLC. DNLC is used within individual file systems; the path cache is used within the VFS lookup code only. Also, the results of `path-lookup` cannot be used to fill entries in DNLC, since DNLC maps component to vnode, whereas the path cache maps path to vnode.

The `path-lookup` call should be directed only to servers that are capable of handling it. The proper approach would be to alter the MOUNT protocol so that, at mount time, the file server indicates if it can handle `path-lookup`, and, if so, which types of pathname syntax it understands. The client would then store this information in

the `struct vfs` for that mount. However, to reduce the number of required protocol changes, our code assumes that every mounted file system understands the call, and tries it. If a “bad operation” RPC error occurs, or if the RPC succeeds but the server indicates that it cannot handle the syntax of the pathname, the server’s inability is recorded in the `struct vfs`. Besides avoiding a change to the MOUNT protocol, this approach has the advantage of slightly easing incremental deployment. A disadvantage is that a server’s limits are repeatedly re-discovered (once per mount), and automounters — which are increasingly common — tend to enormously increase the number of times that a file system is (un)mounted.

2.2 Details

This section explains how path lookup adjusts to the three special cases: symlinks, mount points, and dot-dot.

2.2.1 Symlinks

Any component of a path may be a symlink, and symlinks may expand to anything. Therefore, the servers and directories visited while translating a path are not predictable simply by inspecting the initial path.

For an example, consider the path “`./x/y/z`” illustrated in Figure 1. If `x` were a mount point, then `y/z` should be resolved in a different file system than it would be if `x` were not a mount point. The client could detect if `x` were a mount point, since the client knows its mount points. However, `x` could also be a symlink that would expand to `w`, which in turn may or may not be a mount point, leading to the same predicament.

The catch-22 is that the client cannot know which server to contact until it knows whether the path “is what it seems to be” and the client cannot know that a path is what it seems to be until its components have been looked up at the server.

To break the cycle, we optimistically assume that the path contains no special cases. Referring to the example above, the client would lookup the path `x/y/z` on the server for directory “.” (which is necessarily the right server for the lookup of `x`). The server responds with a partition of `x/y/z` into a symlink-free prefix and a suffix that begins with the expansion of the first symlink, if any exists.

The reason that the `path-lookup` operation translates only to the first symlink and not to the end of the path is that the optimistic assumption may be false. If the path does contain a special case, the server is probably wasting some effort translating a path that is different from the one

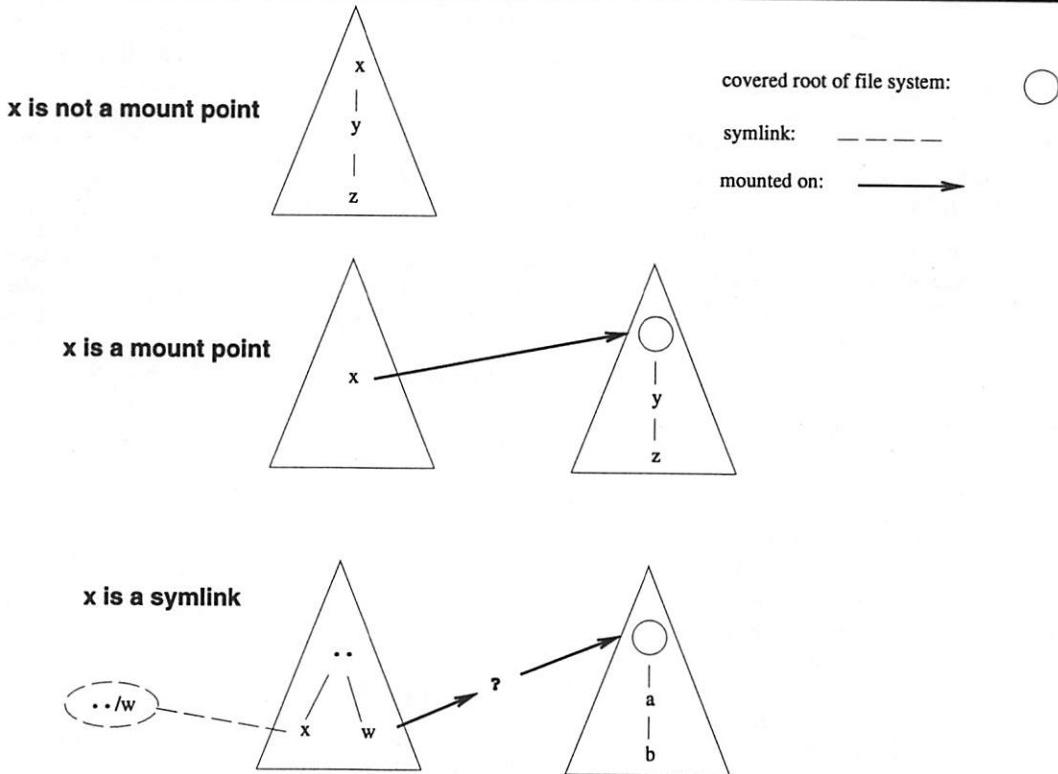


Figure 1: Uncertainty Caused By Symlinks

that should be translated. In such cases, the client must have enough information to survive the false assumption. We chose to have the server return from the *path-lookup* as soon as it encounters information that might signal that the optimistic assumption is false. Note that the server cannot return when the path crosses a mount point because the mount points that are relevant are those on the client and, practically speaking, the server cannot know the paths of the client's mount points. So the server is doing all it can by returning when it encounters a symlink. Having the server return on every symlink has essentially no effect on performance (because of the rarity of special cases) and somewhat simplifies the client (since the client need retain information for and check for only two of the three special cases).

2.2.2 Mount Points

After the *path-lookup* returns, the client will examine the symlink-free prefix for mount points. If no mount point is found, then the prefix was translated on the correct server. So the algorithm repeats by sending the suffix, if any, to the same server. If the path of some mount point is contained in the prefix, then the path lookup may

have been directed to the wrong server: so the portion of the path (prefix and suffix) below the first mount point is sent to the server for the mounted file system (assuming that it understands *path-lookup*).

The reason for the mount-point check is that a server that looks up a path does so with respect to *its* name space; however, the semantics of file name translation demand that a path be translated with respect to the name space of the client. Consider the example in Figure 2. During the translation of `/usr/local-gnu/bin/emacs`, the name `gnu/bin/emacs` is translated by Server A because the client has mounted that server's file system on its name `/usr/local`. However, the client has also mounted Server B's file system on the name `/usr/local-gnu`. Therefore, the correct translation is that of `bin/emacs` with respect to Server B's file system, rather than `gnu/bin/emacs` with respect to Server A's file system. So the translation provided by Server A may be wrong.

In order to have enough information to check for mount points, the client accumulates the symlink-free prefixes returned from all *path-lookup* calls. After each call returns, the current accumulated symlink-free prefix is compared against all mount

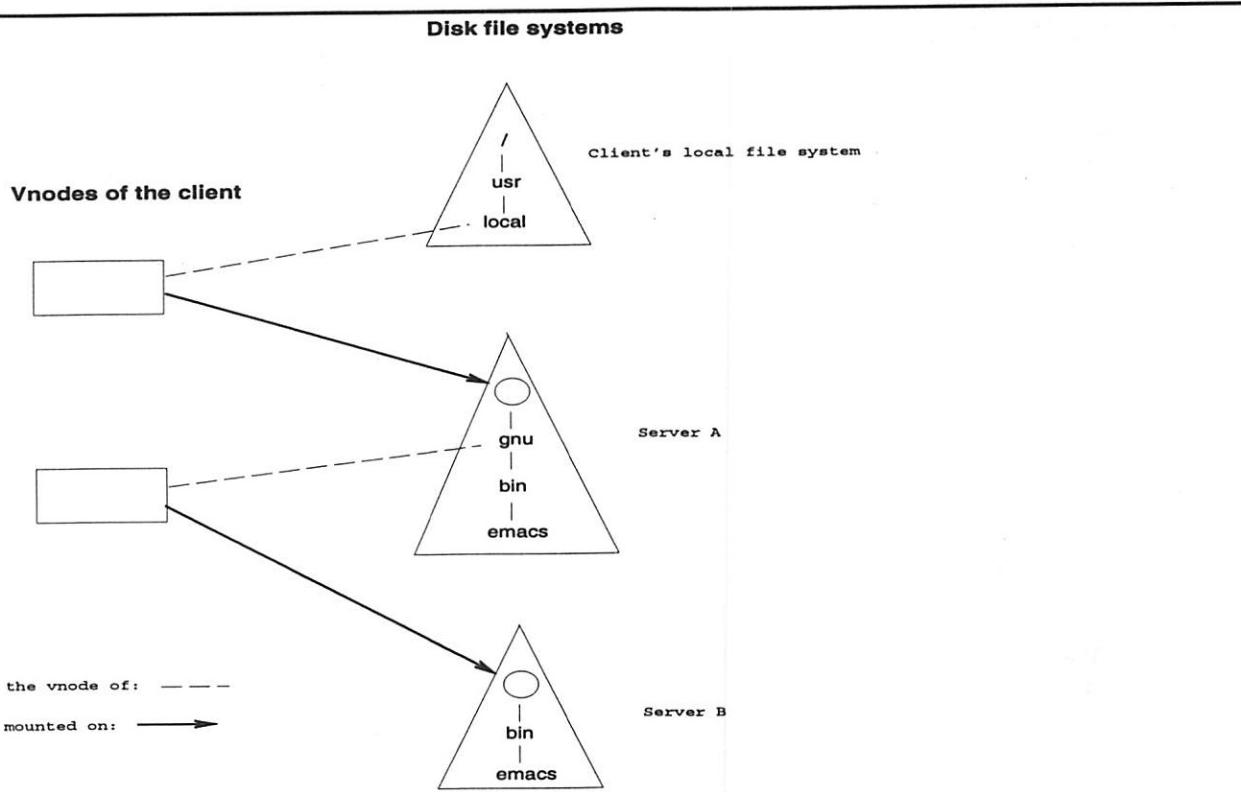


Figure 2: Why Symlink-free Path Must Be Compared Against Mount Points

points. In order to provide fast search through all mount points, we added a trie index that points to all NFS mount points. The trie stores absolute path names, as shown in Figure 3. However, the pathnames generated by a process are resolved relative to either its current root (curroot) or current working directory (cwd). This twist presents no problem: the vnodes for cwd and curroot are available, and each contains a pointer to its **struct vfs**; by definition, these structures are represented in the trie by their complete, absolute pathnames. Therefore, a pathname lies in a file system different from the one housing the starting point iff:

1. There is another mount point farther down the branch of the trie housing the **struct vfs** of the starting point.
2. The pathname is not embedded in the trie between the **struct vfs** of the starting point and the next **struct vfs**.

Our implementation platform, SunOS 4.1.3, keeps the path of all its mount points only in the file **/etc/mtab**. For three reasons, we made changes so that the name of a mount point is also kept in the associated **struct vfs**. First, for performance: the pathnames of mount points have

to be accessed on every path lookup. Second, to avoid race conditions: after initiating an I/O to access **/etc/mtab** the kernel would continue; the kernel's next operation might be another that access or manipulates **/etc/mtab**. Finally, as part of earlier work [10], we had already written some code to store mount point names in **struct vfs**.

2.2.3 Dot-dot

Dot-dot must be handled with care similar to that for mount points. The reason is the same: the server will interpret dot-dot with respect to its name space, whereas the required semantics are with respect to the client's name space. Usually the two interpretations are the same. The only exception is if dot-dots in the path result in going above the root of the remote file system.⁶ For example, suppose **/usr/local** is an exported file system; then the path **"/usr/local/.."** refers to **/usr** on the *client*, not the server.

Unfortunately, we thought of no simple and clean check for and adjustment to the possibility of backing up over the root of the contain-

⁶The NFS server checks for and prevents this case on every lookup.

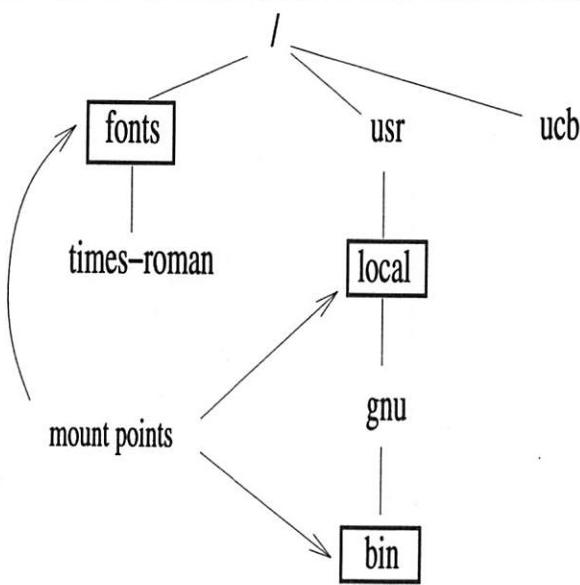


Figure 3: Trie Index for Comparing a Path Against All Mount Points

ing file system. The rub is that it is messy for the client to remember the path of every starting point (i.e., process cwd or curroot). So before each call to `path-lookup`, the client checks if the path goes above *the starting point* (i.e., cwd or curroot) of the translation. If so, the path lookup is aborted and the regular VFS lookup algorithm is used. This conservative approach means that any path that begins with dot-dot is not processed using `path-lookup`. Similarly, after the final `path-lookup` operation responds, the client checks the symlink-free prefix; if the prefix would back up over the starting point, then the path lookup algorithm is aborted.

2.2.4 Path Cache

The path cache is referenced from within the new VFS lookup algorithm and from within the NFS code for validating caches. Most of the code for the path cache was copied and adapted from DNLC.

Since NFS provides no means for a server to call back to a client, the path cache can contain outdated information.⁷ Stale cache entries are removed by NFS's normal timeout-driven checking of vnode attributes and, since the cache is managed LRU, by aging the oldest entry during an insert operation. All these traits are shared with DNLC.

⁷The same is true for DNLC and, indeed, for a client-side cache of any type of information about remote NFS files.

One difference between the path cache and DNLC is that explicit deletion (such as when a file is deleted or renamed) is handled slightly differently. The delete and rename operations delete from the path cache by vnode since a vnode is a unique ID and since it would be difficult to construct a path for the target. DNLC entries can be deleted by either vnode or component name.

2.2.5 Protocol Change

The definition of the new call added to version 2 of the NFS protocol is:

```

struct pathlookupargs {
    nfspath pathname;
    int syntax;
};

struct pathlookupokres {
    nfs_fh file;
    fattr attributes;
    nfspath prefix;
    nfspath suffix;
};

union pathlookupres
switch (nfsstat status) {
case NFS_OK:
    pathlookupokres pathlookupres;
default:
    void;
};

pathlookupres
NFSPROC_PATH_LOOKUP(pathlookupargs) = 18;
  
```

A new “unintelligible syntax” error code was necessary. However, since, at the server, path lookup is simply an iterative application of the regular `lookup` operation, all the same access constraints apply.

2.2.6 Server Change

To implement the `path-lookup` operation on the server side, we stole code from other NFS operations, especially `lookup`. Essentially, the `path-lookup` implementation is that of `lookup` with two main additions:

1. Instead of translating a single component, the code iterates over components until it reaches the end, a symlink, or an error.
2. When a symlink is encountered, it is read (by calling the server-side operation to read a symlink) and prepended to the remaining untranslated path.

3 Evaluation

This algorithm is implemented in SunOS 4.1.3 and is part of the operating system regularly booted on nine SparcStations. Fewer than a thousand lines of code were added. On the server side, only a small addition was made to the module that implements NFS operations (`nfs_server.c`). Most changes were on the client side, where some modules received major changes: `vfs_lookup.c`, and a few others needed to store the path name of mount points in the `struct vfs`.

Before implementing, we studied the distribution of lengths of pathnames. The longer the pathname given to the path lookup algorithm, the greater the upside potential. Measuring in the same environment as noted earlier — eight major multi-user departmental file servers — we found considerable variation in path length distribution from machine to machine and time to time. However, on average path lengths of 4 were most common, with lengths of 3 the next most common. Paths of length 3 or 4 together accounted for over 70% of lookups. Of the remainder, most were shorter.

To measure the effect of our changes, we used a kernel build as a benchmark; all major kernel sources, libraries, and include files were on a remote file system. Within a short period of time, a kernel build opens a large number of files in a relatively small number of directories. Kernel builds provide a friendly test for path lookup: the average path length is somewhat longer (4.6) than the more comprehensive number noted above. Because of background activity, the results varied a little, but on average the build ran 8% faster with path lookup in effect. Eight percent is a substantial speedup considering that it is the effect of changing *only* the lookup operation.

Measuring the effect path lookup has on the server is at once easier and harder than measuring client latency. Measuring number of server operations is easy, but each path lookup operation can be expected to perform more work than an ordinary NFS `lookup`. We used `nfsstat` to measure the number of operations serviced and `vmstat` to record processor idle time and I/O operations. For a set of kernel build benchmarks, the number of NFS operations declined 20% and processor idle time averaged 16% higher with path lookup in effect. Apparently, processor overhead for handling NFS requests is substantial.

3.1 Violation of NFS Design Principle

As noted earlier, the addition of path-lookup violates the longstanding design decision to keep the

NFS protocol free of path syntax.

One possible rejoinder is that, while it is true that it is desirable to keep operating system specifics out of the NFS protocol, this design decision was made several years ago; since then, NFS, though widely ported, has received almost all its use on DOS and UNIX platforms. We question whether the substantial negative impact on performance caused by component-by-component lookup is acceptable considering that the abstraction offered by omitting pathname syntax “abstracts” over effectively only two implementations.

Another, possibly better, rejoinder would be to re-design our protocol change so that it accepted and returned not paths, but rather vectors of opaque components. It would still be necessary to exchange an indication of how to interpret the components, but at least the letter, if not the spirit, of the original design principle would be preserved. We have not yet made this change to the protocol.

4 Related Work

As noted in the introduction, `lookup`, `getattr`, and `null` comprise the vast majority (over 80%) of NFS operations handled by our main servers. The high number of `null` operations is attributable to our use of the Amd automounter [5]; Amd periodically “pings” every mounted file system, and NFS `null` is the ping operation. While the high number of `null` operations can thus be dismissed as site-specific, the dominance of `getattr` and `lookup` is typical for most NFS installations — as the operation mix in `nhfsstone` indicates. So naturally there has been interest in sharply reducing the frequency of these operations.

Most such interest has focused on `getattr`, although in most experiments the motivation has not been simply to reduce the frequency of `getattr` but rather to improve the consistency guarantee provided to the client by an NFS server. One early experiment is “Spritely NFS” [6], in which a callback scheme similar to that in Sprite [4] was added to NFS with the intention of providing strict cache consistency and improving performance by eliminating the overhead of refreshing the attribute cache with `getattr`. The ideas in Spritely NFS were used in modified form in “NQNFS” [3] which is a second protocol available from the NFS implementation of 4.4BSD. NQNFS differs from Spritely NFS in that the latter requires the server to keep state indicating the cache status of files at clients; however, NQNFS borrows the “lease” idea from Gray and Cheriton [2] in order to avoid the need for servers to keep state across failures. In NQNFS, a client is allowed to

cache a file for a specified period of time. The only recovery action a server must take is to wait until such time as all of its leases must have expired. Neither of these systems addresses the issue of reducing `lookups`.

More recently, the specification for version 3 of NFS [8] included, among its many changes, the requirement to return attributes as a side effect of every appropriate NFS operation. The intention is to reduce the number of separate `getattr` operations that must be invoked in order to verify attribute cache consistency.

Cache consistency and NFS version 3 are a bit far afield, but we are not aware of any attempts — besides DNLC — to reduce the cost and/or number of NFS `lookup` operations. However, the notion of looking up and caching whole or partial paths has been proposed before for new file system designs [9, 1].

In 1986 Welch and Ousterhout described “prefix tables” [9]. Prefix tables are useful in an environment where a shared global hierarchy of files is partitioned into “domains,” which are spread across servers. Each client maintains a prefix table that maps file name prefixes to the servers on which the associated domains reside. Prefix table entries are hints: if a file is not where a table says it is, shorter prefixes are tried until the file is found. If a client has no prefixes at all for a file (as will be the case initially), it broadcasts the file name to all servers. Relevant prefix/server mappings are returned by all servers that have such mappings. In this way, prefix table information can be easily propagated without the requirement that any two clients have precisely the same table. And because prefix tables contain only hints that need not be correct, this method avoids creating either an availability or a consistency problem.

The prefix table idea is quite similar to our work; however, there is one major difference between the model of file system use in NFS and that in the prefix table proposal. Welch and Ousterhout describe a construct called a “remote link,” which is apparently a replacement for the idea of client mounts. Distinct domains are stitched together with remote links, which are server-side mounts; that is, the client has no control over how to overlay domains on top of one another — the information is encoded in remote links in the file system, and all clients see the same arrangement of domains into a hierarchy. Implementing static mounts on the server side is a significant simplification for whole-path translation (and a significant loss of flexibility for the client). In our design, the server returns after encountering a symlink and the client must check the symlink-free path for mount points — both of these features exist only

because the server cannot know the client’s mount points. In Welch and Ousterhout’s system, unlike in NFS, there is no complication with having the server expand a symlink and continue translating it without contacting the client. In their work, the only time a server returns a pathname to the client not completely translated is when some component of the path crosses a domain boundary: either dot-dot in the upward direction or a remote link in the downward direction. In these cases, the client must check its prefix table to learn which server to send the remainder of the path to. In summary, prefix tables is an elegant idea but one targeted for a significantly different and easier model of file system definition and use.

In [1], Cheriton and Mann describe a naming system that is scalable enough to encompass the world and general enough to name many types of objects (not just files — processes, windows, network connections, etc.) Since the features that draw most of their design attention are those that permit scaling to enormous size, it is hard to make a meaningful comparison between our work and theirs. Their system has the notion of looking up and caching whole or partial pathnames. However, they reject the notion of client mounts on the grounds that such client-specific name space management operations do not scale well and stand in the way of forming a consistent global name space. The notion of symbolic links seems absent from their design, presumably on similar grounds.

5 Summary

Measurements of NFS pathnames and `lookup` performance yield several pronounced facts:

- The hit rate of DNLC is not easily improved; nevertheless, enough `lookup` operations cannot be satisfied from DNLC so that `lookup` is the operation that most commonly goes over the wire to the server.
- Average path length is long enough so that translating a completely uncached path will often require as many as 3 or 4 `lookup` operations.
- Paths given to the VFS lookup algorithm almost never contain symlinks or cross mount points, and so can typically be translated at the server in a single operation.

Given these facts, one may question whether the elegance and relative simplicity of the VFS lookup algorithm — which translates uncached pathnames one component at a time — is sufficient compensation for its high overhead.

Indeed, the server load caused by repeated NFS lookup operations can be reduced by up to 16% if path lookup is used instead of component lookup. Also, path lookup can have a noticeable effect on client latency for workloads that are open-intensive.

Path lookup is not hard to implement, the only tricky aspect being that a translated path must be examined even after a “successful” translation in order to ensure that the middle of the translated path did not cross a mount point. If it did, then the pathname looked up at the server is not the path that should have been looked up — the portion of the pathname below the mount point has different meaning on the two different servers.

6 Acknowledgements

Andreas Prodromidis helped gather many of the reported statistics. Margo Seltzer suggested making the arguments/results of the path-lookup call be vectors of opaque components.

This work was supported in part by ONR grant number N00014-93-1-0315, and by National Science Foundation CISE Institutional Infrastructure grant number CDA-90-24735.

References

- [1] D. R. Cheriton and T. P. Mann.
Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance.
ACM Trans. Computer Systems, 7(2):147–183, May 1989.
- [2] C. G. Gray and D. R. Cheriton.
Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency.
In *Proc. Twelfth ACM Symp. Operating Systems Principles*, pages 202–210, December 1989.
- [3] R. Macklem.
Not Quite NFS, Soft Cache Consistency for NFS.
In *Proc. 1994 Winter USENIX*, pages 261–278, January 1994.
- [4] M. N. Nelson, B. B. Welch, and J. K. Ousterhout.
Caching in the Sprite Network File System.
ACM Trans. Computer Systems, 6(1):134–154, February 1988.
- [5] J. Pendry and N. Williams.
Amd – The 4.4 BSD Automounter.
Imperial College of Science, Technology, and Medicine, London, 5.3 alpha edition, March 1991.
- [6] V. Srinivasan and J. C. Mogul.
Spritley NFS: Implementation and Performance of Cache-Consistency Protocols.
Research report 89/5, DEC Western Research Lab, May 1989.
- [7] Sun Microsystems, Inc.
NFS: Network File System Protocol Specification.
RFC 1094, IETF Network Working Group, March 1989.
- [8] Sun Microsystems, Inc.
NFS: Network File System Version 3 Protocol Specification.
June 25, 1993.
Available from gatekeeper.dec.com:/pub/standards/nfs/nfsv3.ps.Z
- [9] B. Welch and J. Ousterhout.
Prefix Tables: A Simple Mechanism for Locating Files in a Distributed System.
In *Sixth Intl. Conf. Distributed Computing Systems*, pages 184–189, May 1986.
- [10] E. Zadok and D. Duchamp.
Discovery and Hot Replacement of Replicated Read-Only File Systems, with Application to Mobile Computing.
In *Proc. 1993 Summer USENIX*, pages 69–85, June 1993.

Dan Duchamp is an Associate Professor of Computer Science at Columbia University. His current research interest is the various issues in mobile computing. For his initial efforts in this area, he has been named an Office of Naval Research Young Investigator.

Mail address: Computer Science Department, Columbia University, 500 West 120th Street, New York, NY 10027. Email address: djd@cs.columbia.edu.

Application-Controlled File Caching Policies

Pei Cao, Edward W. Felten, and Kai Li

Department of Computer Science
Princeton University
Princeton, NJ 08544 USA

Abstract

We consider how to improve the performance of file caching by allowing user-level control over file cache replacement decisions. We use two-level cache management: the kernel allocates physical pages to individual applications (*allocation*), and each application is responsible for deciding how to use its physical pages (*replacement*). Previous work on two-level memory management has focused on replacement, largely ignoring allocation.

The main contribution of this paper is our solution to the allocation problem. Our solution allows processes to manage their own cache blocks, while at the same time maintains the dynamic allocation of cache blocks among processes. Our solution makes sure that good user-level policies can improve the file cache hit ratios of the entire system over the existing replacement approach. We evaluate our scheme by trace-based simulation, demonstrating that it leads to significant improvements in hit ratios for a variety of applications.

1 Introduction

File caching is a widely used technique in today's file system implementations. Since CPU speed and memory density have improved dramatically in the last decade while disk access latency has improved slowly, file caching has become increasingly important. One major challenge in file caching is to provide high cache hit ratio.

This paper studies an application-controlled file caching approach that allows each user process to use an application-tailored cache replacement policy instead of always using a global Least-Recently-Used (LRU) policy. Some applications have special knowledge about their file access patterns which can be used to make intelligent cache replacement decisions. For example, if an application knows which

blocks it needs and which it does not, it can keep the former in cache and reduce its cache miss ratio.

Traditionally such applications buffer file data in user address space as a way of controlling replacement. However, since the kernel tries to cache file data as well, this approach leads to double buffering, which wastes space. Furthermore, this approach does not give applications real control because the virtual memory system can still page out data in the user address space. Hence we need another way to let applications control replacement.

To reduce the miss ratio, a user-level file cache needs not only an application-tailored replacement policy but also enough available cache blocks. In a multiprocess environment, the allocation of cache blocks to processes will thus affect the file cache hit ratio of the entire system. It is the kernel's job to ensure that the hit ratio of the whole system does not degrade because of the user-level management of cache replacement policies. The challenge is to allow each user process to control its own caching and at the same time to maintain the dynamic allocation of cache blocks among processes in a fair way so that overall system performance improves.

This paper describes a scheme that achieves this goal. Our approach, called "two-level block replacement", splits the responsibilities of allocation and replacement between kernel and user level. A key element in this scheme is a sound allocation policy for the kernel, which is discussed in section 3. This allocation policy guarantees that an application-tailored replacement policy can improve the overall file system performance and that a foolish replacement policy in one application will not degrade the file cache hit ratios of other processes.

We have evaluated our allocation policy using trace-driven simulation. In our simulations, we used several file access traces that we collected on a DEC 5000/200 workstation running the Ultrix operating system, and the Sprite traces from University of California at Berkeley. We have simu-

lated our allocation policy and various replacement policies for individual application processes. The simulations show that an application-tailored replacement policy can reduce an application's file cache miss ratio up to 100%, over the global LRU policy. In addition, in a multiprocess environment, the combination of our allocation policy and application-tailored replacement policies can reduce the overall file cache miss ratios, over the traditional global file caching approach, by up to 50%.

2 User Level File Caching

Our goal is to allow user-level control over cache replacement policy. In many cases, the application has better knowledge about its future file accesses than the kernel has. User level control of cache replacement enables the application to use its better knowledge to improve the hit ratio.

Despite the advantages of application control, we cannot simply move all responsibility for cache management to the user level. In a multiprogrammed system, the kernel serves a valuable function: managing the allocation of resources between users to guarantee the performance of the entire system.

2.1 Two-Level Replacement

We propose a scheme for file caching that splits responsibility between the kernel and user levels. The kernel is responsible for allocating cache blocks to processes. Each user process is free to control the replacement strategy on its share of cache blocks; if it chooses not to exercise this choice, the kernel applies a default policy (LRU). We call our approach *two-level cache block replacement*.

To be more precise, each file is assigned to a "manager" process, which is responsible for making replacement decisions concerning the file. Usually the process that currently has the file open is its manager; however, this process may designate another process to be the manager of a particular file. If several processes have the same file open simultaneously, then it is up to these processes to agree on a manager; if they cannot agree then the kernel imposes the default LRU policy for that file. Processes that do not want to control their own replacement policy can abdicate their management responsibility; in this case the kernel applies the default LRU policy for the affected files.

The interactions between kernel and manager processes are the following: On a cache miss, the kernel finds a candidate block to replace, based on

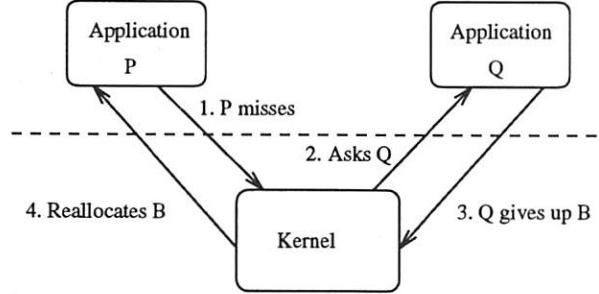


Figure 1: Interaction between kernel and user processes in two-level replacement: (1) P misses; (2) kernel consults Q for replacement; (3) Q decides to give up page B; (4) kernel reallocates B to P.

its global replacement policy (step 1 in Figure 1). The kernel then identifies the manager process of the candidate. This manager process is given a chance to decide on the replacement (step 2). The candidate block is given as a hint, but the manager process may overrule the kernel's choice by suggesting an alternative block under that manager's control (step 3). Finally, the block suggested by the manager process is replaced by the kernel (step 4). (If the manager process is not cooperative, then the kernel simply replaces the candidate block.)

The kernel's replacement policy is in fact an allocation policy. Suppose that process P's reference misses in the cache and the kernel finds a replacement candidate owned by process Q. Although process Q's user-level replacement policy decides *which* of its blocks will be replaced, the replacement will cause a deallocation of a block from process Q and an allocation of a block to process P.

2.2 Kernel Allocation Policy

The kernel allocation policy is the most critical part of two-level replacement. To obtain best performance, it is known that allocation should follow the *dynamic partition* principle [11]: each process should be allocated a number of cache blocks that varies dynamically in accordance with its working set size. Experience has shown that global LRU or its approximations perform relatively well; they approximate the dynamic partition principle or tend to follow processes' working set sizes.

Our goal is to design an allocation policy for the kernel to guarantee that the two-level replacement method indeed improves system performance over the traditional global (or single level) replacement method when user processes are not making bad replacement decisions. To be more precise, let us call a decision to overrule the kernel *wise* if the alter-

native block is referenced before the candidate suggested by the kernel, and call the decision *foolish* if the candidate will be referenced first. (A decision not to overrule the kernel can be viewed as neutral.) The kernel's allocation policy should satisfy three principles:

1. *A process that never overrules the kernel does not suffer more misses than it would under global LRU.* This ensures that ordinary processes, which are unwilling or unable to predict their own accesses, will perform at least as well as they did before.
2. *A foolish decision by one process never causes another process to suffer more misses.* Of course, we cannot prevent a process from discarding its valuable pages. However, we must ensure that an errant or malicious process cannot hurt the performance of others.
3. *A wise decision by one process never causes any process, including itself, to suffer more misses.* This ensures that processes have an incentive to choose wisely. (It goes without saying that wise decisions should actually improve performance whenever possible.)

The main contribution of this paper is to propose and evaluate an allocation policy that satisfies these design principles.

3 An Allocation Policy

We will describe our allocation policy in an evolutionary fashion. We will start with a simple but flawed policy, and then diagnose and fix two problems with it. The result will be a fully satisfactory allocation policy.

3.1 First Try

To start, we can have an allocation policy that is literally the same as that of global LRU. The kernel simply maintains an LRU list of all blocks currently in the file cache. When a replacement is necessary, the block at the end of the LRU list is suggested as a candidate, and its owner process is asked to give up a block.

The problem is that if the owner process overrules the kernel, the candidate block still stays at the end of the LRU list. On the next miss, the same process will again be asked to give up a block.

The left side of Figure 2 shows an example. Two processes, P and Q, share a file cache with four blocks. Process P uses blocks A and B; process Q

uses blocks W, X, Y, and Z. The reference stream is Y, Z, A, B.

The top line shows the initial LRU list, at time t_0 . The first reference, to Y at time t_1 , causes a replacement. The kernel consults its LRU list and suggests A for replacement. A's owner, process P, overrules the kernel. It decides to replace B, hoping to save the next miss to A. The LRU list is now as shown at time t_2 . The next reference, to Z at time t_3 , causes another replacement. Again, A is chosen as a candidate. This time P has no other blocks in the cache and hence must give up A. The LRU list is now as shown at time t_4 . At this point, the next two references, to A and B, both miss.

There are four misses in this example, two misses by P and two by Q. But note that under global LRU, there would be only three misses, one by P and two by Q. This violates Principle 3: a wise decision by process P causes P to suffer one extra miss.

3.2 Swapping Position

The problem in the above scheme arises because the LRU list is maintained in strict reference order. Intuitively, the only use of the LRU list is to decide which process will give up a block upon a miss. To get the same allocation policy as the existing global LRU policy, our policy's LRU list should be in correspondence to the LRU list in the original algorithm. This can be achieved by swapping the blocks' positions in the LRU list.

Suppose the kernel suggests A for replacement, but the user-level manager overrules it with B. At this point, the previous policy would simply replace B. The new policy first swaps the positions of A and B in the LRU list, then proceeds to replace B. As a result of this swap, A is no longer at the tail of the LRU list. Compared with the LRU list under global LRU (i.e. if A is replaced), the only difference is that A is in B's position.

This fixes the problem with above example as in Figure 2. The right side of the figure shows what happens under the new policy. On the first replacement, A moves to the head of the LRU list before B is replaced. The result is that A is still in the cache when it is referenced. Process P is no longer hurt by its wise choice.

In general, swapping positions guarantees that if no process makes foolish choices, the global hit ratio is the same as or better than it would be under global LRU.

Unfortunately, this scheme does not guard against foolish choices made by user processes. This is illustrated by the left side of Figure 3. Processes

		First try method		Swapping position	
Time	P's ref Q's ref	Cache State	Global LRU list	Cache state	Global LRU list
t_0		A W X B	$A \rightarrow W \rightarrow X \rightarrow B$	A W X B	$A \rightarrow W \rightarrow X \rightarrow B$
t_1	Y	A W X B ↓↑	$A \rightarrow W \rightarrow X \rightarrow Y$	A W X Y	$W \rightarrow X \rightarrow A \rightarrow Y$
t_2	Z	A W X Y	$W \rightarrow X \rightarrow Y \rightarrow Z$	A Z X Y	$X \rightarrow A \rightarrow Y \rightarrow Z$
t_3		A Z X Y ↓↑	$X \rightarrow Y \rightarrow Z \rightarrow A$	A Z X Y	$X \rightarrow Y \rightarrow Z \rightarrow A$
t_4	A	Z W X Y	$Y \rightarrow Z \rightarrow A \rightarrow B$	A Z B Y	$Y \rightarrow Z \rightarrow A \rightarrow B$
t_5		Z A X Y			
t_6	B	Z A B Y			
t_7					
t_8					

Figure 2: This example shows what's wrong with the first try and how to fix it with the swapping position mechanism.

P and Q share a three-block cache. The top line shows the initial LRU list at time t_0 ; the reference stream is Z, Y, A. The first reference, to Z at time t_1 , causes a replacement. The kernel suggests X for replacement. Now suppose process Q makes exactly the wrong choice: it decides to replace Y. After swapping X and Y in the LRU list, the kernel replaces Y, leading to the LRU list as shown at time t_2 . The second reference, to Y at time t_3 , misses. The kernel suggests A for replacement, and process P cannot overrule because it has no alternative to suggest. Thus A is replaced, leading to the LRU list as shown at time t_4 . The third reference, to A at time t_5 , misses.

There are three misses in this example, one miss by P and two by Q. Under global LRU, there would be only one miss, by Q. Had Q not foolishly overruled the kernel, the last two references would both have hit in cache. Principle 2 is violated — process Q's foolish decision causes process P to suffer an extra miss.

3.3 Place-Holders

The problem in the previous example arises because Q's choice enables it to acquire more cache blocks than it would have had under global LRU. As a result, some of P's blocks are pushed out of the cache, which increases P's miss rate. To satisfy Principle 2, we must prevent foolish processes from acquiring extra cache blocks. We achieve this by using *place-holders*.

A place-holder is a record that refers to a page. It records which block would have occupied that page under the global LRU policy. Suppose the kernel

suggests A for replacement, and the user process overrules it and decides to replace B instead. In addition to swapping the positions of A and B in the LRU list, the kernel also builds a place-holder for B to point to A's page. If B is later referenced before A, A's page can be confiscated immediately. This allows the cache state to recover from the user process's mistake.

The right side of Figure 3 illustrates how place-holders work. The top line shows the initial LRU list at time t_0 . The first reference, to Z at time t_1 , misses. Block X is chosen as a candidate for replacement, but process Q (foolishly) overrules the kernel and chooses Y for replacement. X and Y are swapped in the LRU list, and Y is replaced. At this point, a place-holder is created, denoting the fact that the block occupied by X would have contained Y under global LRU.

The second reference, to Y at time t_3 , misses. The kernel notices that there is a place-holder for Y — Y "should" have been in the cache, but was not, due to a foolish replacement decision by Y's owner. The kernel responds to this situation by correcting the foolish decision: it loads Y into the page that Y's place-holder pointed to, replacing X. Note that in this case the normal replacement mechanism is bypassed.

The LRU list is now as shown at time t_4 . The third reference, to A, hits.

This example results in two misses, both by process Q. Under global LRU, there would have been only one miss, by Q. Q hurts itself by its foolish decision, but it does not hurt anyone else. Principle 2 is satisfied.

Time			No Place-Holder		With Place-Holder	
	P's ref	Q's ref	Cache State	Global LRU list	Cache state	Global LRU list
t_0	Z	A	X A Y ↓↑	$X \rightarrow A \rightarrow Y$	X A Y ↓↑	$X \rightarrow A \rightarrow Y$
t_1			X A Z ↓↑	$A \rightarrow X \rightarrow Z$	X A Z ↓↑	$A \rightarrow X(Y) \rightarrow Z$
t_2			X Y Z ↓↑	$X \rightarrow Z \rightarrow Y$	Y A Z ↓↑	$A \rightarrow Z \rightarrow Y$
t_3			X Y Z ↓↑	$Z \rightarrow Y \rightarrow A$	Y A Z ↓↑	$Z \rightarrow Y \rightarrow A$
t_4						
t_5						
t_6						

Figure 3: This example shows why place-holders are necessary.

3.4 Our Allocation Scheme

Combining above two fixes, here is our full allocation scheme: If a reference to cache block b hits, then b is moved to the head of the global LRU list, and the place-holder pointing to b (if there is one) is deleted. If the reference misses, then there are two cases. In the first case, there is a place-holder for b , pointing to t ; in this case t is replaced and its page is given to b . (If t is dirty, it is written to disk.)

In the second case, there is no place-holder for b . In this case, the kernel finds the block at the end of the LRU list. Say that block c , belonging to process P , is at the end of the LRU list. The kernel consults P to choose a block to replace. (The kernel suggests replacing c .) Say that P 's choice is to replace block x . The kernel then swaps x and c in the LRU list. If there is place-holder pointing to x , it is changed to point to c ; otherwise a place-holder is built for x , pointing to c . Finally, x 's page is given to b . (x is written to disk if it is dirty.)

We can prove that this algorithm satisfies all three of our design principles. (A detailed formal proof appears in [4].) Our scheme has the property that it never asks a process to replace a block for another process more often than global LRU. In other words, whenever a process is asked to give up a block for another process, it would have already given up that block under global LRU.

To see why, first notice that the place-holder scheme ensures that every process *appears*, in the view of other processes, never to unwisely overrule the kernel's suggestions. This is because whenever a process makes an unwise decision, only the errant process is punished and the state is restored as if the mistake were never made.

Hence, from the allocator's point of view, every

process is either doing LRU or is doing something better than LRU. Therefore we need only guarantee that those that are doing better than LRU are not discriminated against. Swapping position serves this purpose. That is, the allocator doesn't care which of a process's pages is holding which data, as long as its pages occupy the same positions on the LRU list that they would have occupied under global LRU.

In summary, our framework for incorporating user level control into replacement policy is: consulting user processes at the time of replacement; swapping the positions of the block chosen by the global policy and the block chosen by the user process in the LRU list; and building "place-holder" records to detect and recover from user mistakes. This framework is also applicable to various policies that approximate LRU, such as FIFO with second chance, and two-hand clock[16].

4 Design Issues

This section addresses various aspects of our scheme, including possible implementation mechanisms, treatment of shared files and interaction with prefetching.

4.1 User-Kernel Interaction

There are several ways to implement two-level replacement, trading off generality and flexibility versus performance.

The simplest implementation is to allow each user process to give the kernel hints. For example, a user process can tell the kernel which blocks it no longer needs, or which blocks are less important than others. Or it can tell the kernel its access pattern for some file (sequential, random, etc). The

kernel can then make replacement decisions for the user process using these hints.

Alternatively, a fixed set of replacement policies can be implemented in the kernel and the user process can choose from this menu. Examples of such replacement policies include: LRU with relative weights, MRU (most recently used), LRU-K[21], etc.

For full flexibility, the kernel can make an upcall to the manager process every time a replacement decision is needed, as in [18].

Similarly, each manager process can maintain a list of “free” blocks, and the kernel can take blocks off the list when it needs them. The manager would be awakened both periodically and when its free-list falls below an agreed-upon low-water mark. This is similar to what is implemented in [25].

Combinations of these schemes are possible too. For example, the kernel can implement some common policies, and rely on upcalls for applications that do not want to use the common policies. In short, all these implementations are possible for our two-level scheme. We are still investigating which is best.

4.2 Shared Files

As discussed in Section 2.1, concurrently shared files are handled in one of two ways. If all of the sharing processes agree to designate a single process as manager for the shared file, then the kernel allows this. However, if the sharing processes fail to agree, management reverts to the kernel and the default global LRU policy is used.

4.3 Prefetching

Under two-level replacement, prefetches could be treated in the same way as in most current file systems: as ordinary asynchronous reads.

Most file systems do some kind of prefetching. They either detect sequential access patterns and prefetch the next block[17], or do cluster I/O[19].

Recent research has explored how to prefetch much more aggressively. In this case, a significant resource allocation problem arises — how much of the available memory should the system allocate for prefetching? Allocating too little space diminishes the value of prefetching, while allocating too much hurts the performance of non-prefetch accesses. The prefetching system must decide how aggressively to prefetch.

Our techniques do not address this problem, nor do they make it worse. The kernel prefetching

code would still be responsible for deciding how aggressively to prefetch. We would simply treat the prefetcher as another process competing for memory in the file cache. However, since the prefetcher would be trusted to decide how much memory to use, our allocation code would provide it with a fresh page whenever it wanted one.

Recent research on prefetching focuses on obtaining information about future file references[22]. This information might be as valuable to the replacement code as it is to the prefetcher, as we discuss in the next section. Thus, adding prefetching may well make the allocator’s job easier rather than harder.

To facilitate the use of a sophisticated prefetcher, there can be more interaction between the allocator and the prefetcher. For example, the allocator could inform the prefetcher about the current demand for cache blocks; the prefetcher could voluntarily free cache blocks when it realized some prefetched blocks were no longer useful, etc. The details are beyond the scope of this paper.

5 Simulation

We used trace-driven simulation to evaluate two-level replacement. In our simulations the user-level managers used a general replacement strategy that takes advantage of knowledge about applications’ file references. Two sets of traces were used to evaluate the scheme.

5.1 Simulated Application Policies

Our two-level block replacement enables each user process to use its own replacement policy. This solves the problem for those sophisticated applications that know exactly what replacement policy they want. However, for less sophisticated applications, is there anything better than local LRU? The answer is yes, because it is often easy to obtain knowledge about an application’s file accesses, and such knowledge can be used in replacement policy.

Knowledge about file accesses can often be obtained through general heuristics, or from the compiler or application writer. Here are some examples:

- Files are mostly accessed sequentially; the suffix of a file name can be used to guess the usage pattern of a file: “.o” files are mostly accessed in certain sequences, “.ps” files are accessed sequentially and probably do not need to be cached, etc.

- Compilers may be able to detect whether there is any `lseek` call to a file; if there is none, then it is very likely that the file is accessed sequentially. Compilers can also generate the list of future file accesses in some cases; for example, current work on TIP (Transparent Informed Prefetching)[22] is directly applicable.
- The programmer can give hints about the access pattern for a file: sequential, with a stride, random, etc.

When these techniques give the exact sequence of future references, the manager process can apply the offline optimal policy RMIN: replace the block whose next reference is farthest in the future. Often, however, only incomplete knowledge about the future reference stream is known. For example, it might be known that each file is accessed sequentially, but there might be no information on the relative ordering between accesses to different files. RMIN is not directly applicable in these cases. However, the principle of RMIN still applies.

We propose the following replacement policy to exploit partial knowledge of the future file access sequence: when the kernel suggests a candidate replacement block to the manager process,

1. find all blocks whose next references are definitely (or with high probability) after the next reference to the candidate block;
2. if there is no such block, replace the candidate block;
3. else, choose the block whose reference is farthest from the next reference of the candidate block.

Depending on the implementation of two-level replacement, this policy may be implemented in the kernel or in a run-time I/O library. Either way, the programmer or compiler needs to predict future sequences of file references.

This strategy can be applied to common file reference patterns. For general applications, common file access patterns include:

- *sequential*: Most files are accessed sequentially most of the time;
- *file-specific sequences*: some files are mostly accessed in one of a few sequences. For example, object files are associated with two sequences: 1) sequential; 2) first symbol table, then text and data (used in link editing);

- *filter*: many applications access files one by one in the order of their names in the command line, and access each file sequentially from beginning to end. General filter utilities such as grep are representative of such applications;
- *same-order*: a file or a group of files are repeatedly accessed in the same order. For example, if “*” (for file name expansion) appears more than once in a shell script, it usually leads to such an access pattern; and
- *access-once*: many programs do not reread or rewrite file data that they have already accessed.

Applying our general replacement strategy, we can determine replacement policies for applications with these specific access patterns. Suppose the kernel suggests a block A, of file F, to be replaced. For *sequential* or *file-specific*, the block of F that will be referenced farthest in the future is chosen for replacement; for *filter*, the sequence of future references are known exactly, and RMIN can be applied; for *same-order*, the most recently accessed block can be replaced; and for *access-once*, any block of which the process has referenced all the data can be replaced.

5.2 Simulation Environment

We used trace-driven simulations to do a preliminary evaluation of our ideas. Our traces are from two sources. We collected the first set ourselves, tracing various applications running on a DEC 5000/200 workstation. The other set is from the Sprite file system traces from University of California at Berkeley[2].

We built a trace-driven simulator to simulate the behavior of the file cache under various replacement policies¹. In our simulation we only considered accesses to regular files — accesses to directories were ignored for simplicity, the justification being that file systems often have a separate caching mechanism for directory entries. We also assume that the file system has a fixed size file cache², with a block size of 8K.

We validated our simulator using Ultrix traces. Our machine has a 1.6MB file cache. We can measure the actual number of read misses using the Ultrix “time” command. Our simulation results were

¹Our traces and simulator are available via anonymous ftp from [ftp.cs.princeton.edu: pub/pc](ftp://ftp.cs.princeton.edu/pub/pc).

²That is, we do not simulate the dynamic reallocation of physical memory between virtual memory and file system that happens in some systems.

within 3% of the real result except for link-editing, for which the simulator predicted 7% fewer misses. This is because the simulator ignores directory operations, which are more common in the link-editing application.

To evaluate our scheme, we compared it with two policies: existing kernel-level global LRU without application control, and the ideal offline optimal replacement algorithm, RMIN. The former is used by most file systems, while the latter sets an upper bound on how much miss ratio can be reduced by improving the replacement policy. Our performance criterion is miss ratio: the ratio of total file cache misses to total file accesses.

5.3 Results for Ultrix Traces

We instrumented the Ultrix 4.3 kernel to collect our first set of traces. When tracing is turned on, file I/O system calls from every process are recorded in a log, which is later fed to the simulator.

Traces were gathered for three application programs, both when they were running separately and when they were run concurrently. The applications are:

- *Postgres*: Postgres is a relational database system developed at University of California at Berkeley[28]. We used version 4.1. We traced the system running a benchmark from the University of Wisconsin, which is included in the release package. The benchmark takes about fifteen minutes on our workstation.
- *cscope*: cscope is an interactive tool for examining C sources. It first builds a database of all source files, then uses the database to answer the user's queries, such as locating all the places a function is called. We traced cscope when it was being used to examine the source code for our kernel. The trace recorded four queries, taking about two minutes.
- *link-editing*: The Ultrix linker is known as being I/O-bound. We collected the I/O traces when link-editing an operating system kernel twice, taking about six minutes. We also collected traces when linking some programs with the X11 library.

Single Application Traces First we'd like to see how introducing application control can improve each application's caching performance:

- *Postgres*: it is often hard to predict future file accesses in database systems. To see whether

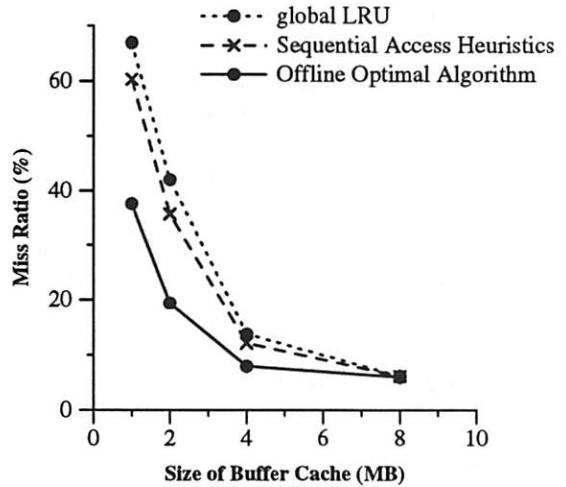


Figure 4: Performance for Postgres

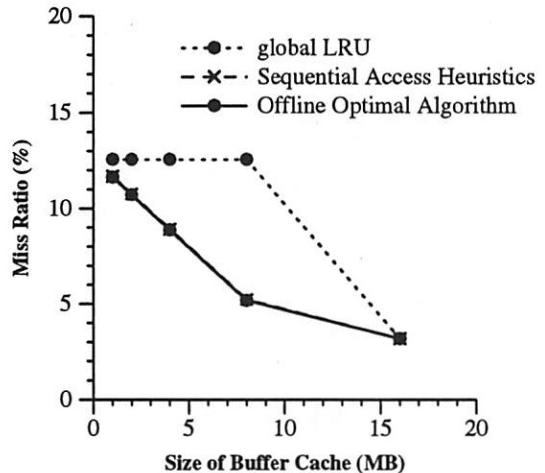


Figure 5: Performance for cscope

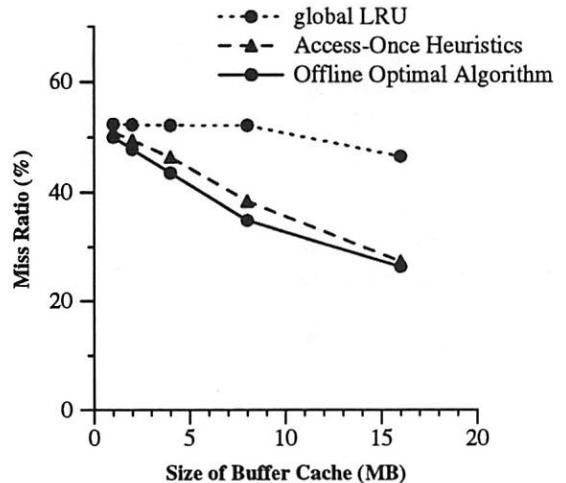


Figure 6: Performance for Linking Kernel

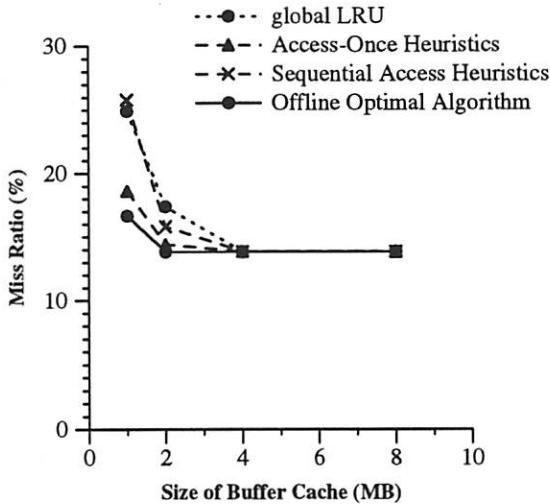


Figure 7: Performance for Linking with X11

user-level heuristics may reduce miss ratio, we tried the policy for *sequential* access pattern. Indeed, the miss ratio is reduced (Figure 4). We think that the designer of the database system can certainly give a better user-level policy, thus further improving the hit ratio.

- *cscope*: *cscope* actually has a very simple access pattern. It reads the database file sequentially from beginning to end to answer each query. The database file used in our trace is about 10MB. For caches smaller than 10MB, LRU is useless. The reason that the miss ratio is only 12.5% is that the size of file accesses is 1KB, while the file block size is 8KB. However, if we apply the right user-level policy (noticing that the access pattern is *same-order*), the miss ratio is reduced significantly (Figure 5).
- *link-editing*: the linker in our system makes a lot of small file accesses. It doesn't fit the *sequential* access pattern. However, it is *read-once*. Even though the linker is run twice in our traces, during each run its user level policy can still be that of *read-once*. The result is shown in Figure 6. For linking with X11 library, we tried both the policy for *sequential* and the policy for *read-once* at user-level (Figure 7). *read-once* seems to be the right policy. (Note that this trace is small and a 4MB cache is actually enough for it.)

Multi-Process Traces Having seen that appropriate user-level policies can really improve the cache performance of individual applications, we

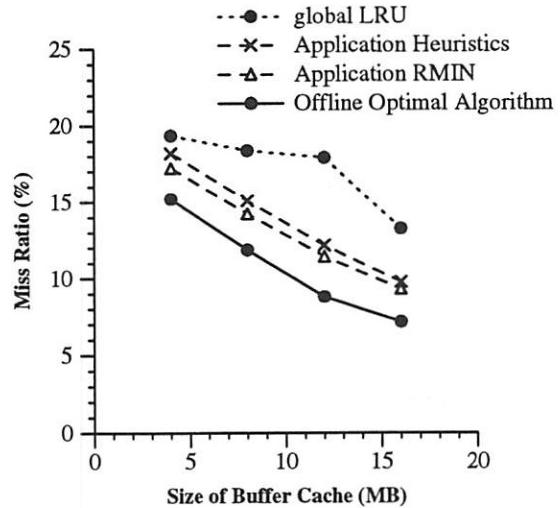


Figure 8: Performance for a Multi-Process Workload

would like to see how our scheme performs in a multi-process environment.

We collected traces when the three applications (Postgres, cscope, linking the kernel) are run concurrently. In this trace, we simulated each application running its own user-level policy as discussed above. The result is shown in Fig.8. Since the applications' user-level policies are not optimal, we also simulate the case of each application using an offline optimal algorithm as its user-level policy. This yields the curve directly above RMIN.

As can be seen, our scheme, coupled with appropriate user level policies, can improve the hit ratio for multiprocess workloads.

We also performed an experiment to measure the benefit of using place-holders. We collected a trace of the Postgres and kernel-linking applications running concurrently, and simulated the miss ratio of Postgres when kernel-linking makes the worst possible replacement choices and Postgres simply follows LRU. We simulated our full allocation algorithm and our algorithm without place-holders. Figure 9 shows the result. Without place-holders, Postgres is noticeably hurt by the other application's bad replacement decisions. (With place-holders, Postgres has the same miss ratio as under global LRU.)

5.4 Results for Sprite Traces

Our second set of traces is from the UC Berkeley Sprite File System Traces [2]. There are five sets of traces, recording about 40 clients' file activities over a period of 48 hours (traces 1, 2 and 3) or 24 hours (traces 4 and 5).

We focused on the performance of client caching.

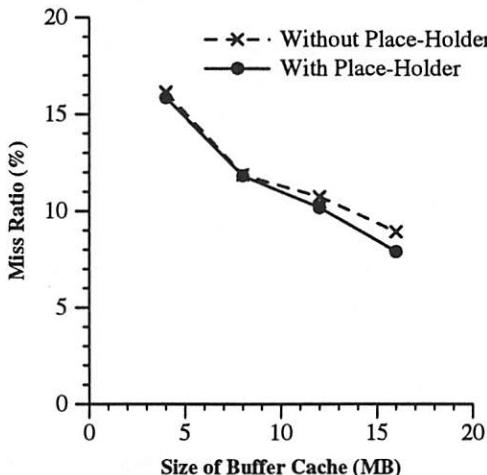


Figure 9: Benefit of Using Place-Holders

In a system with a slow network (e.g. ethernet), client caching performance determines the file system performance on each workstation. Furthermore we ignored kernel file activities in these traces, because Sprite's VM system swaps to remote files. In our simulation we set the client cache size to be 7MB, which is the average file cache size reported in [2].

These traces do not contain process-ID information, so we cannot simulate application-specific policies as with Ultrix traces. However, since most file accesses are sequential [2], the *sequential* heuristic can be used. Figure 10 shows average miss ratios for global LRU, sequential heuristic and optimal replacement. Average cold-start (compulsory) miss ratios are also shown.

As can be seen, two-level replacement with sequential heuristic improves hit ratio for some traces. In fact simulations show that *sequential* improves hit ratio for about 10% of the clients, and the improvements in these cases are between 10% and over 100%.

Overall, these results show that two-level replacement is a promising scheme to enable application control of replacement and to improve the hit ratio of the file buffer cache. We believe that two-level replacement should be implemented in future file systems.

6 Related Work

There have been many studies on caching in file systems (e.g. [24, 5, 23, 3, 13, 20]), but these investigations were not primarily concerned with cache replacement policies. The database community has long studied buffer replacement policies[26, 8, 21],

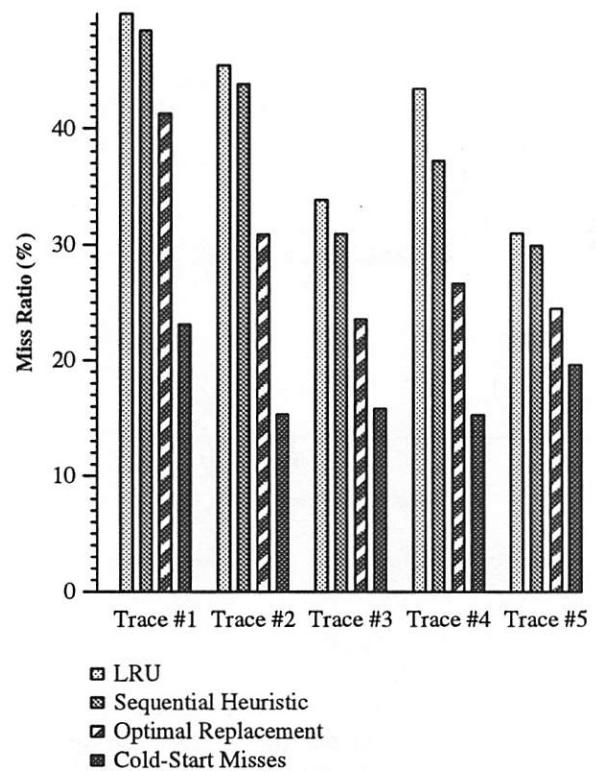


Figure 10: Averaged Results from Sprite Traces

but existing file systems do not support them. Although user scripts were introduced in caching in a disconnected environment[14], they were used to tell the file system which files should be cached (on disk) when disconnection occurs. In most of these systems, the underlying replacement policy is still LRU or an approximation to it.

In the past few years, there has been a stream of research papers on mechanisms to implement virtual memory paging at user level. The external pager in Mach [29] and V [6] allows users to implement paging between local memory and secondary storage, but it does not allow users to control the page replacement *policy*. Several studies [18, 25, 12, 15] proposed extensions to the external pager or improved mechanisms to allow users to control page replacement policy. These schemes do not provide resource allocation policies that satisfy our design principles to guarantee replacement performance. Furthermore, they are not concerned with file caching.

Previous research on user-level virtual memory page replacement policies [1, 9, 12, 15, 27] shows that application-tailored replacement policies can improve performance significantly. With certain modifications, these user-level policies might be used as user-level file caching policies in our

two-level replacement. Recent work on prefetching [22, 10, 7] can be directly applied in user-level file caching policies using our general replacement strategy. These systems can take advantage of the properties of our allocation policy to guarantee performance of the entire system.

7 Conclusions

This paper has proposed a two-level replacement scheme for file cache management, its kernel policy for cache block allocation, and several user-level replacement policies. We evaluated these policies using trace-driven simulation.

Our kernel allocation policy for the two-level replacement method guarantees performance improvements over the traditional global LRU file caching approach. Our method guarantees that processes that are unwilling or unable to predict their file access patterns will perform at least as well as they did under the traditional global LRU policy. Our method also guarantees that a process that mis-predicts its file access patterns cannot cause other processes to suffer more misses. Our key contribution is the *guarantee* that a good user-level policy will improve the file cache hit ratios of the entire system.

We proposed several user-level policies for common file access patterns. Our trace-driven simulation shows that they can improve file cache hit ratios significantly. Our simulation of a multiprogrammed workload confirms that two-level replacement indeed improves the file cache hit ratios of the entire system.

We believe that the kernel allocation policy proposed in this paper can also be applied to other instances of two-level management of storage resources. For example, with small modifications, it can be applied to user-level virtual memory management.

Although the kernel allocation policy guarantees performance improvement over the traditional global LRU replacement policy, there is still room for improvement. We plan to investigate these possible improvements and implement the two-level replacement method to evaluate our approach with various workloads.

Acknowledgments

We are grateful to our paper shepherd Keith Bostic and the USENIX program committee reviewers for their advice on improving this paper. Chris Maeda, Hugo Patterson and Daniel Stodolsky provided helpful comments. We benefited from

discussions with Rafael Alonso, Matt Blaze, and Neal Young. Finally we'd like to thank the Sprite group at the University of California at Berkeley for collecting and distributing their file system traces.

References

- [1] Rafael Alonso and Andrew W. Appel. An Advisor for Flexible Working Sets. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 153–162, May 1990.
- [2] Mary Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John Ousterhout. Measurements of a Distributed File System. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pages 198–211, October 1991.
- [3] Andrew Braunstein, Mark Riley, and John Wilkes. Improving the efficiency of UNIX file buffer caches. In *Proceedings of the Eleventh Symposium on Operating Systems Principles*, pages 71–82, 1989.
- [4] Pei Cao. Analysis of Two-Level Block Replacement. Working Paper, January 1994.
- [5] David Cheriton. Effective Use of Large RAM Diskless Workstations with the V Virtual Memory System. Technical report, Dept. of Computer Science, Stanford University, 1987.
- [6] David R. Cheriton. The Unified Management of Memory in the V Distributed System. Draft, 1988.
- [7] Khien-Mien Chew, A. Jyothy Reddy, Theodore H. Romer, and Avi Silberschatz. Kernel Support for Recoverable-Persistent Virtual Memory. Technical Report TR-93-06, University of Texas at Austin, Dept. of Computer Science, February 1993.
- [8] Hong-Tai Chou and David J. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *Proceedings of the Eleventh International Conference on Very Large Databases*, pages 127–141, August 1985.
- [9] Eric Cooper, Scott Nettles, and Indira Subramanian. Improving the Performance of SML Garbage Collection using Application-Specific Virtual Memory Management. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, June 1992.
- [10] Kenneth M. Curewitz, P. Krishnan, and Jeffrey Scott Vitter. Practical Prefetching via Data Compression. In *Proc. 1993 ACM-SIGMOD Conference on Management of Data*, pages 257–266, May 1993.
- [11] Edward G. Coffman, Jr. and Peter J. Denning. *Operating Systems Theory*. Prentice-Hall, Inc., 1973.
- [12] Kieran Harty and David R. Cheriton. Application-Controlled Physical Memory using External Page-Cache Management. In *The Fifth International*

- Conference on Architectural Support for Programming Languages and Operating Systems*, pages 187–197, October 1992.
- [13] John H. Howard, Michael Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, pages 6(1):51–81, February 1988.
 - [14] James J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, pages 6(1):1–25, February 1992.
 - [15] Keith Krueger, David Loftness, Amin Vahdat, and Tom Anderson. Tools for the Development of Application-Specific Virtual Memory Management. In *OOPSLA '93 Conference Proceedings*, pages 48–64, October 1993.
 - [16] H. Levy and P. Lipman. Virtual Memory Management in the VAX/VMS Operating System. *IEEE Computer*, pages 35–41, March 1982.
 - [17] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, August 1984.
 - [18] Dylan McNamee and Katherine Armstrong. Extending The Mach External Pager Interface To Accommodate User-Level Page Replacement Policies. In *Proceedings of the USENIX Association Mach Workshop*, pages 17–29, 1990.
 - [19] L. W. McVoy and S. R. Kleiman. Extent-like Performance from a UNIX File System. In *1991 Winter USENIX*, pages 33–43, 1991.
 - [20] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite File System. *ACM Transactions on Computer Systems*, pages 6(1):134–154, February 1988.
 - [21] Elizabeth J. O’Neil, Patrick E. O’Neil, and Gerhard Weikum. The LRU-K Page Replacement Algorithm For Database Disk Buffering. In *Proc. 1993 ACM-SIGMOD Conference on Management of Data*, pages 297–306, May 1993.
 - [22] Hugo Patterson, Garth Gibson, and M. Satyanarayanan. Transparent Informed Prefetching. *ACM Operating Systems Review*, pages 21–34, April 1993.
 - [23] R. Sandberg, D. Boldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Summer Usenix Conference Proceedings*, pages 119–130, June 1985.
 - [24] M. D. Schroeder, D. K. Gifford, and R. M. Needham. A caching file system for a programmer’s workstation. *ACM Operating Systems Review*, pages 19(5):35–50, December 1985.
 - [25] Stuart Sechrest and Yoonho Park. User-Level Physical Memory Management for Mach. In *Proceedings of the USENIX Mach Symposium*, pages 189–199, 1991.
 - [26] Michael Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, v. 24, no. 7, pages 412–418, July 1981.
 - [27] Indira Subramanian. Managing discardable pages with an external pager. In *Proceedings of the USENIX Mach Symposium*, pages 77–85, October 1990.
 - [28] The Postgres Group. POSTGRES Version 4.1 Release Notes. Technical report, Electronics Research Lab, University of California, Berkeley, February 1993.
 - [29] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the Eleventh Symposium on Operating Systems Principles*, pages 63–76, November 1987.

Author Information

Pei Cao graduated from TsingHua University in 1990 with a BS in Computer Science. She received her MS in Computer Science from Princeton University in 1992 and is currently a Ph.D. candidate. Her interests are in operating systems, storage management and parallel systems. Email address: pc@cs.princeton.edu.

Edward W. Felten received his Ph.D. degree from the University of Washington in 1993 and is currently Assistant Professor of Computer Science at Princeton University. His research interests include parallel and distributed computing, operating systems, and scientific computing. Email address: felten@cs.princeton.edu.

Kai Li received his Ph.D. degree from Yale University in 1986 and is currently an associate professor of the department of computer science, Princeton University. His research interests are in operating systems, computer architecture, fault tolerance, and parallel computing. He is an editor of the IEEE Transactions on Parallel and Distributed Systems, and a member of the editorial board of International Journal of Parallel Programming. Email address: li@cs.princeton.edu.

Resolving File Conflicts in the Ficus File System*

Peter Reiher, John Heidemann, David Ratner, Greg Skinner, Gerald Popek

Department of Computer Science

University of California, Los Angeles

Abstract

Ficus is a flexible replication facility with optimistic concurrency control designed to span a wide range of scales and network environments. Optimistic concurrency control provides rapid local access and high availability of files for update in the face of disconnection, at the cost of occasional conflicts that are only discovered when the system is reconnected. Ficus reliably detects all possible conflicts. Many conflicts can be automatically resolved by recognizing the file type and understanding the file's semantics. This paper describes experiences with conflicts and automatic conflict resolution in Ficus. It presents data on the frequency and character of conflicts in our environment. This paper also describes how semantically knowledgeable resolvers are designed and implemented, and discusses our experiences with their strengths and limitations. We conclude from our experience that optimistic concurrency works well in at least one realistic environment, conflicts are rare, and a large proportion of those conflicts that do occur can be automatically solved without human intervention.

1 Introduction

The value of file replication is widely recognized, but replication of updatable files leads immediately to consistency problems. File replicas can be partitioned from each other for a variety of reasons, ranging from failures of machines and networks to intentionally intermittent connections (e.g., connection via modem, or replicas on portable machines that are not always attached to a network). If no efforts are taken, partition-

ing can permit conflicting updates to different replicas of a file. Much of the value of replication is based on all replicas being identical, so inconsistent updates are a potentially serious problem.

Early solutions to the problem relied on various conservative algorithms that prevented conflicting updates to different replicas [1]. These solutions used a wide variety of mechanisms, but their common theme is that they refuse updates that have any possibility of causing conflicting updates. These solutions trade availability for consistency. When consistency of replicas is of vital importance, conservative solutions are preferable.

However, experience with file access by typical users has shown that many files are only accessed by a single user [10]. Of those that are shared by multiple users, few are updated by more than one user. In such environments, a mechanism that prevents one user from updating a file in favor of preserving the update ability of other users who might never generate an update is seriously flawed. Conservative replication mechanisms exhibit this flaw.

Optimistic replication mechanisms do not. They allow any replica of a file to be updated at any time. This choice ensures that users who need to update a file can do so when any replica is available. However, optimistic mechanisms gain this availability by trading off consistency. Since any replica can be updated, two non-communicating replicas can be changed independently, leading to *conflicts*, i.e., different replica contents. To maintain consistency, a system with optimistic replication must detect and recover from such conflicts.

The improved availability of optimistic systems must be weighed against the frequency and cost of recovering from conflicts. A hypothesis of this paper is that the cost of optimism is low in many environments.

To test this hypothesis, this paper reports conflict resolution experiences with Ficus, an optimistic file system developed at UCLA [4]. Ficus has supported

*This work was sponsored by the Defense Advanced Research Projects Agency under contract N00174-91-C-0107. Gerald Popek is also affiliated with Locus Computing Corporation.

The authors can be reached at 4760 Boelter Hall, UCLA, Los Angeles, CA, 90024, or by electronic mail to `ficus@cs.ucla.edu`.

the primary computing needs of over a dozen users at UCLA for more than three years.

Ficus has a general architecture for dealing with file conflicts. Conflicts are automatically detected and examined to determine if they can also be resolved automatically. Special programs called *resolvers* handle conflicts that can be dealt with automatically.

Ficus is able to resolve almost all conflicts for a particularly important class of files—directories. Ficus supports a Unix-style directory system; the semantics of Unix directories provide that almost every conflict that can occur in them can be automatically resolved. Directory conflicts could be resolved by submitting them to a resolver that implements the algorithms necessary to resolve their conflicts, but the integrity of the Unix file system is so closely linked to its directories that we have chosen to put the algorithms into Ficus itself.

We have instrumented the Ficus system to keep track of the number of conflicts generated and how many conflicts were resolved automatically. This paper presents the statistics gathered, which support the contention that, for important patterns of usage, the frequency of file conflicts is low enough that optimistic replication is highly attractive. Further, the statistics demonstrate that the use of automatic resolvers is both practical and important to reduce the number of conflicts reported to users.

The next section presents an overview of the Ficus file system. We begin there with an example of how a conflict can arise in an optimistic replication system, discuss the different kinds of conflicts that can arise, briefly describe how conflicts are detected, and cover other relevant aspects of Ficus. Section 3 describes the Ficus resolution architecture. It discusses the automated directory conflict resolution mechanisms in Ficus and describes how Ficus handles other types of conflicts. Section 4 discusses Ficus conflict resolver programs. It covers their interface and the various approaches used to resolve conflicts for different types of files. Section 5 presents conflict data gathered from Ficus; Section 6 discusses some related research. We close with a discussion of future work and some conclusions.

2 Ficus Overview

Ficus is a distributed file system utilizing optimistic replication [16, 4]. The default synchronization policy provides *single copy availability*; so long as any copy of a data item is accessible, it may be updated. Once a single replica has been updated, the system makes a best effort to notify all accessible replicas that a new version of the file exists via update propagation. Those

replicas then pull over the new data. Ficus guarantees *no lost update semantics* despite this optimistic concurrency control. Conflicting updates are guaranteed to be detected, allowing recovery after the fact.

Ficus groups subtrees of files into *volumes*. A volume can be replicated multiple times. A background process known as *reconciliation* runs on behalf of each volume replica after each reboot and periodically during normal operation. It compares all files and directories of the local volume replica with a remote replica of the volume, pulling over missed updates and detecting concurrent update conflicts.

Several types of conflicts are possible. They include:

- Update/update conflicts
- Name conflicts
- Remove/update conflicts

The remainder of this section will discuss these types of conflicts in more detail. The next section describes how we manage these conflicts.

Since single copy availability permits any replica to be updated, even a simple partitioning of a two-replica file can result in a conflict. Figure 1 illustrates this situation. File foo has two replicas in Figure 1a, with replica 1 at site A and replica 2 at site B. If sites A and B are partitioned, as Figure 1b shows, updates to both replicas are accepted. Then, when the partition is merged, as shown in figure 1c, file foo exists in two versions. This is an *update/update conflict*.

Directories provide a special case of update/update conflicts. Partitioned creation of independent files in the same directory would ordinarily result in an update/update conflict on that directory. Since directories are internal to the file system, Ficus automatically resolves this sort of concurrent update, producing the union of all directory changes. (See [6] for a description of the algorithms employed in directory management.) A problem occurs when two files are independently created with the same name; Unix requires that each directory entry be unique. We term this kind of directory update/update conflict a *name conflict*.

Figure 2 illustrates a different kind of conflict. In figure 2a, we see two replicas of file foo before a partition. In 2b, file foo is removed at site B (indicated by the shading of site B's replica), while the partitioned replica at site A is updated. When the partition merges, as shown in 2c, if no update had occurred, then the other replica should simply be removed. However, if the updated replica is removed in this situation, the update generated during the partition is lost, possibly without the knowledge of the person making the update. Ficus' “no lost update semantics” requires that the update generated at site A not be discarded as a result of the

Site A



Site B

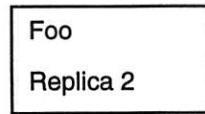


Fig. 1a – Before partition

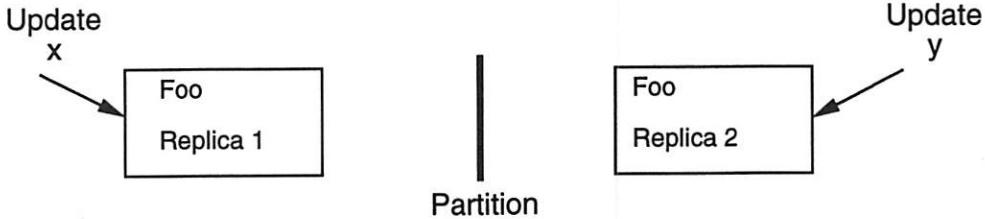


Fig. 1b – During partition



Fig. 1c – After partition

Figure 1: An update/update conflict.

removal of the file at site B. This kind of conflict is called a *remove/update conflict*.

If a file is independently deleted from two replicas of a partitioned directory, Ficus does not log a conflict. This delete/delete situation is not a problem, provided any other replicas of the file are also deleted when the partitions merge, since both deletions have precisely the same effect.

Ficus makes a “best-effort” attempt to propagate updates as they occur. However, even when no partitions or other machine failures happen, update propagation is not guaranteed. Thus, conflicting updates can arise even without machine or network failures. Also, Ficus does not lock replicas for update even within a partition, so two replicas can accept simultaneous updates to a file that could result in a conflict. In practice, the update propagation mechanism is fast and reliable enough that conflicts unrelated to actual failures or partitioning almost never occur.

Ficus detects all types of conflicts using a mechanism known as a *version vector* [14]. Each file replica maintains its own version vector that keeps track of the

history of updates to the file. Conflicts are detected by comparing version vectors from two file replicas. Version vectors reliably detect all file conflicts that involve replicas of a single file. They do not assist in ensuring the consistency of updates that span multiple files. Other mechanisms (not supported in Ficus, nor in most file systems) are required to do so.

3 Ficus Conflict Resolution Architecture

Several types of conflicts are possible in Ficus. Because of the importance of the integrity of directories, directory conflicts receive special handling. Remove/update conflicts also require some special treatment. Update/update conflicts on non-directory files are the most common case. The following subsections discuss each type of conflict and its handling in more detail.

Site A

Site B

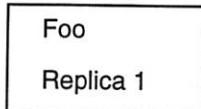


Fig. 2a – Before partition



Fig. 2b – During partition

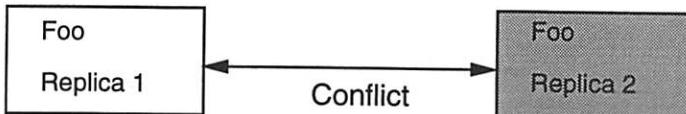


Fig. 2c – After partition

Figure 2: A remove/update conflict.

3.1 Directory Conflicts

The integrity of a Unix file system depends on its directories. If a directory cannot be used because it has received conflicting updates, a portion of the file system's name space may become inaccessible. Thus, conflicts in directories are very serious. Either they must not occur often, or they must be resolved automatically.

Ficus directory conflicts are repaired automatically during reconciliation. As shown in [4, 3], all conflicts that can occur in a Unix directory can be automatically resolved, except for name conflicts. A complete description of directory reconciliation algorithms is available in these references, so we discuss only their broad outlines here, as an example of how known semantics of files can be used to resolve conflicts.

Unix directories support only two operations: a process can add a name to the directory or remove a name from a directory. Creation of a file adds a name to a directory (in addition to creating data structures to represent the file itself). Creating a hard link to a file adds

a second name for the file. File contents are discarded only when the last name for the file has been removed. While mechanically rename is an atomic operation in many Unix systems, semantically it can be treated as a remove followed by a create.

A number of issues, such as handling arbitrary patterns of failures and recoveries, distinguishing between creation and deletion of entries, and avoiding centralized algorithms, make the problem of directory management substantially more complex than it seems at first glance. In broadest principle, the automatic reconciliation mechanisms for directories examine all entries in both versions of the directory in conflict, determine which entries are common to both, and, for an entry that is present in only one directory, determine whether the file was created or deleted during partition. Ficus keeps sufficient information to distinguish precisely the patterns of file entry additions and deletions while partitioned, which in turn allows all possibly conflicting updates to be addressed.

One class of conflicting updates can create another problem, however. Concurrent creation of different

files with the same name results in a directory with two identical, effectively indistinguishable names. Ficus detects that the two files are actually different, but the Unix directory model does not permit different files to have identical names, so some action is required. Ficus appends unique suffixes to each name and invokes name conflict resolvers to handle the situation. Like other conflict resolvers, if automatic resolution fails, a default resolver notifies the file owner, who must either rename or remove one of the files.

3.2 Remove/Update Conflicts

Remove/update conflicts are handled specially. Ficus is able to recognize such cases, again using version vectors. Ficus' "no lost update" semantics requires that an remove/update conflict not result in the loss of the update. On the other hand, all names for the data have been removed, so Ficus should not permit the file to remain available via those names. Ficus' solution to this problem is to move the file into a special directory called an *orphanage*. Each volume has its own orphanage directory located under its root directory. When the reconciliation process moves a file into an orphanage, electronic mail is sent to the owner notifying him, allowing him to decide whether to keep the updated file or discard it.

3.3 Update/Update Conflicts

Some update/update conflicts for non-directory files can be resolved automatically and some cannot, depending on the semantics associated with the file. Ficus has the ability to invoke various resolvers to attempt to handle file conflicts. Ficus allows individual users to specify how they would like their conflicts resolved, but also provides a default system for resolving conflicts when users have not specified their own methods, or when the user's methods fail.

3.4 Conflict Resolution

The Ficus reconciliation process runs through the files in two replicas of a volume, examining each file to determine if it has been modified since the last successful reconciliation. When it discovers conflicting versions of those file replicas, the reconciliation process marks the file as "in conflict." After marking the file, reconciliation invokes resolvers to attempt to fix the conflict. As long as the file is in conflict, normal operations on the file under its usual name will fail. Each replica of the file can be accessed by special mechanisms, and Ficus provides a tool to clear the conflict. Ficus resolver programs must use these capabilities and tools

```
\.newsrcl      /bin/newsrcl_resolver
fortunes\dat   /bin/fortune_resolver
man/cat[1-9]   /bin/man_resolver
\.history     /bin/arbitrary_resolver
\.bash_history /bin/arbitrary_resolver
\.pl$          /bin/prolog_resolver
.*             /bin/default_resolver
```

Figure 3: A resolver selection file. The left column is the pattern to match against the conflicted file's pathname. The right column is the resolver that is invoked in an attempt to fix the conflict.

along with semantic knowledge of particular file types to resolve conflicts.

Ficus selects a resolver to use for a particular conflicted file by searching a personal and a system-wide resolver list. Entries in the system-wide list include resolvers for common file types. Personal resolver lists specify resolvers for file types unique to an individual. Personal resolver lists also allow an individual to choose between safety and convenience by optionally enabling resolvers that don't preserve all data. For example, some users don't care about conflicts on backup files left from editors and so have a resolver arbitrarily select one of conflicting backup files. More conservative users may wish to have data in backup files completely preserved and so may invoke a resolver saving each replica of the backup file as separate files.

Conflict resolvers almost always require knowledge about the type of the file being resolved. Since Unix systems do not provide a typed file system, Ficus infers file types from file names and from type-recognition programs that examine file contents and attributes.

The reconciliation process that examines the resolver lists matches only on file-name-to-regular-expression comparisons. Because regular expressions are used, matches can be exact or on substrings. Figure 3 shows a portion of a resolver file. Whenever a conflicted file named .newsrcl or a file whose pathname contains man/cat is encountered the newsrcl or manual-page resolvers are invoked. All resolvers shown in this figure have been implemented with the exception of the prolog_resolver.

Unfortunately, simple file name comparison cannot reliably identify all file types. For example, the file csh.1 might be a manual page in certain contexts and shell script in others. To support more sophisticated file type identification, a resolver list might use more intelligent programs to check the file type. Continuing with the example in Figure 3, the prolog_resolver could abort if invoked to resolve a Perl program (whose

files end with the same extension), by recognizing that certain constructs in the file were probably not legal Prolog. If a resolver aborts other resolvers are invoked in turn until one succeeds.

Separate resolver lists are provided for name conflicts and file conflicts. In retrospect, the data-specific actions taken in the case of a name conflict and a file conflict are often quite similar; only details about resolving the conflict differ. However, in certain cases it is important for a resolver to know whether it is dealing with a name conflict or an update/update conflict. In the future we plan to merge the different resolver lists and specify the conflict type as an argument to the resolver.

Ficus allows files to be replicated any number of times, so it is possible that a given file might have three or more conflicting replicas. The Ficus reconciliation mechanism works on only two replicas at a time, though, so the conflicting replicas will be dealt with in a pairwise fashion. This simplifies the writing of resolvers, since they need only deal with the common case of exactly two conflicting replicas, rather than an arbitrary number. All conflicts involving multiple replicas can be regarded as multiple pairwise conflicts, so no power is lost. Also, often not all of the conflicting replicas are simultaneously available. Since reconciliation runs between two sites known to be in communication, at least the pair of replicas they store are guaranteed to be available.

How resolvers fix conflicts depends on the semantics of the file in question. Section 4 discusses a variety of the existing Ficus conflict resolvers. Typically, resolvers read the data contained in both versions of the file, update one version of the file on the basis of both versions, then update the version vector of that replica to dominate the other, clearing the conflict.

4 Conflict Resolution Strategies and Examples

Experience with the Ficus conflict resolution mechanism has shown that there are broad classes of file conflicts that can be automatically resolved. This section discusses them, presenting examples of each. We make no claim that the list is exhaustive—in fact, we are sure it is not, since it simply demonstrates the classes of conflicts that have occurred frequently enough in our environment to draw the attention of conflict resolver writers. Further investigation, especially work in different computing environments, will undoubtedly reveal other classes of file conflicts amenable to automatic resolution.

In several important cases, much of the potential

work of resolving conflicts is done by Ficus itself. As mentioned, Ficus resolves most directory conflicts automatically. Thus, any application that makes substantial use of the Unix directory structure has much of its conflict resolution problem automatically solved. Two important examples are Ficus graft points and MH mail directories.

Ficus divides its file space into volumes, each of which is connected to a single place in the file hierarchy. That place is called the volume's *graft point*, similar to a Unix mount point. The graft point must keep information about all the replicas of the volume, including each replica's storage site and other bookkeeping information. Since Ficus uses a Unix directory with one directory entry per graft point entry, graft point conflicts can be resolved automatically. For instance, if a new replica is added on each side of a partition, when the partition merges the graft point will automatically be resolved to indicate that both new replicas are available. Graft points do not experience name conflicts because the tools that update them never generate identical graft point entries.

The MH mail application also makes substantial use of directories. In MH, messages are organized into *folders*, which are implemented as directories. Most of the conflicts that could occur to MH folders during a partition are thus resolved automatically. For example, if a user re-files mail messages into different folders on both sides of a partition, the Ficus directory conflict resolution mechanism would handle most of the resulting conflicts. Only name conflicts occasionally caused by re-filing messages into the same position in two replicas of a given folder require user attention. If numeric identity of messages is not considered important, even these conflicts can be automatically resolved.

Another type of conflict that Ficus resolves automatically is conflicts on files whose contents can be automatically reconstructed. Control files used by the MH mail system are an example. These files maintain sequence and context mechanisms. They can, and often are, built as needed by MH. The only requirement is that MH generally expects something to be there—it is not prepared to deal with a totally absent file, though it can deal with a file that does not contain very useful information. Thus, to resolve conflicts on these files, the file contents are truncated. The next time MH is run, the file will be reconstructed with a default context.

Many types of files are not important to most users. For example, many users do not care about core files produced by Unix processes that fail, or about backup files produced by some Unix programs. Users who do not care about such files can put lines in their personal resolver files that either delete all such files when they get in conflict, or choose one of the conflicting replicas

arbitrarily, or choose the replica with the later date. However, since some users do not want to lose some of their core or backup files without their knowledge, the system resolver file does not impose these decisions on users.

Some files are monotonically increasing logs of information. An example is the `.newsrc` file listing what newsgroups and articles have been read. The message numbers listed as read in each newsgroup usually increase monotonically. In the case of truly monotonically increasing logs, resolving conflicts is simple. The post-resolution version of the conflicting file simply contains the high water mark for each entry. If the file keeps exhaustive lists of items, the resolved version merges items from both conflicting versions.

In the particular case of `.newsrc` files, the situation is a little more subtle. The semantics of what can be changed in a `.newsrc` file is a bit richer than simply updating a record of articles seen. The user can subscribe or unsubscribe to newsgroups, for example. Some of these actions remove information from the file, making perfect conflict resolution impossible. The Ficus conflict resolver for `.newsrc` files thus must make some choices. It generally errs on the side of information preservation, presenting users with more news rather than with less. For instance, if one version of a `.newsrc` file indicates that a newsgroup is not subscribed to, and the other version indicates that it is, the conflict resolver subscribes the user to that newsgroup. The user can easily unsubscribe again, if that was truly what he wanted to do. If the system had left him unsubscribed when he had just recently subscribed, however, the user might not notice that his subscription had been invisibly revoked. This is an example of the create/delete ambiguity described in [5]. In some cases, taking one possible action and reporting the action taken to the user may be sufficient.

The `.newsrc` resolver shows a typical characteristic of many resolvers. Often, it is relatively easy to produce a resolver that is right the great majority of the time, but occasionally makes a mistake. Producing one that is right all of the time, on the other hand, may be very difficult, or even impossible. A reasonable strategy in such cases is to write a resolver anyway, as long as the resolver can do something in the tricky cases that will not produce disastrous results. If the results are merely inconvenient in the rare cases when they're not necessarily right, then the resolver has solved the conflict correctly most of the time, and caused little more trouble than not solving it at all the rest of the time. Since the alternative to this choice is notifying the user to solve it himself, this approach is attractive.

In some cases, the semantics of a file are quite simple. Score files for some of the popular Unix games are

one such case. These files typically keep the top scores in sorted order. Ficus has conflict resolvers for many such games that sort and merge the two conflicting versions, removing duplicates. The case of game score files does bring up a difficulty with writing general resolvers, however. While each of the game score files contains substantially the same kind of information, the actual format is sufficiently different that writing a single resolver to handle all of them is difficult. Instead, Ficus has a class of very similar resolvers to deal with the peculiarities of each.

In other cases, conflicts can be solved simply by merging the two versions of a file into one, preserving all data in both. Doing so may cause some data to be duplicated, but many programs are able to handle such duplications without problems. One such case is the `xcal` program, an interactive window-based calendar manager. Conflicted `xcal` data files can be resolved simply by concatenating the two versions into one. The Ficus resolver includes a comment line indicating what happened, should the user care to clean up further, but the `xcal` program can go ahead and work with the merged version.

When the Ficus resolver files cannot resolve a conflict themselves, they call a final resolver (called the *generic resolver*) that notifies the user via electronic mail that a conflict occurred. The conflict is left unresolved until the user gives it personal attention.

In the UCLA environment, every replica of a volume is reconciled with another replica every hour. In the case of unresolvable conflicts, users might be bombarded with hourly messages about unresolved conflicts that hadn't been fixed. If a user did not log in over a weekend, fifty or more messages could accumulate in his mailbox telling him about a single conflict.

This problem is prevented by keeping track of unresolved conflicts that have been brought to the owner's attention already. When the generic resolver sends out a message about a conflict, it also logs the conflict in a per-volume conflict log file. The next time the resolver notices this conflict, it also reads the entry in the conflict log and determines that it need not send out another message to the user. The conflict log successfully limits the number of conflict report messages users receive.

However, since the conflict log is replicated in its volume (for very good reasons), this log itself can experience conflicts. Therefore, the conflict log itself needs a conflict resolver. This resolver is another example of how one can easily write a resolution mechanism that is correct almost all of the time, even though it is hard to write one that is always absolutely correct. The conflict log resolver must make sure that a given conflict is reported only once, but also that all conflicts

are reported. It does so by reading both versions and writing a new version that contains all lines in either conflicted version. In case of any problems that cannot be automatically solved, the conflict log resolver simply removes both conflicting versions. Should that happen, the next time a conflict is detected a new conflict log will be created. The user will receive another message for each unresolved conflict in the volume, but no conflicts go unreported and only one extra message per conflict is sent.

Most of the existing Ficus conflict resolvers are written in Perl. Some are written in C, and some in other scripting languages. Generally, resolvers can be written in any language, provided they accept the parameters that the reconciliation process passes to them, and they return a value indicating success or failure. So far, the processing a typical resolver must perform has proven particularly suitable to Perl, in that resolvers frequently perform pattern matching, sorting, and merging, all functions that are provided by Perl. In most cases, the files in question are small, so the greater processing speed C could provide is not important.

There are undoubtedly many other types of files whose conflicts can be resolved automatically. Our approach is to first write resolvers for several known problems areas, then to write resolvers for conflicts that actually come up in practice. Thus, the set of resolvers used at UCLA gradually grows as new types of files generate conflicts and people tire of solving the conflicts by hand. At the moment, we have about 15 different resolvers, some of which are used to resolve multiple file types.

5 Data on Conflict Occurrence and Resolution

The Ficus file system has been running as the primary development environment for Ficus itself for several years. Recently, we began to gather data about the occurrence of conflicts in Ficus. This data was gathered by logging every conflict detected by the reconciliation processes and tracking conflict resolution. In addition to recording conflict detection and resolution, we also recorded the total number of updates made to determine the relative frequency of conflicting updates in our environment.

One shortcoming of this data is that most independent directory updates are *not* detected by this instrumentation. We detect all name conflicts, but do not detect the much more common case of independent creation of two differently named files. Such update "conflicts" are automatically resolved by Ficus direc-

tory resolution algorithms. We know that many such cases have arisen and have required this automatic resolution code. For example, many programs create temporary files in a user's home directory. Such programs would have created many conflicting directory updates between home-use and office machines were it not for automatic directory reconciliation. Unfortunately, this sort of conflict is not represented in our statistics, and we currently cannot precisely estimate the frequency of this situation.

The nature of the environment has a strong influence on how often conflicting updates will occur. An environment in which almost all the machines are connected almost all the time will generate relatively few conflicts. An environment in which some machines are often disconnected will generate more conflicts.

The UCLA environment contains approximately a dozen Sun workstations, each with a regular user, sharing a replicated namespace over an Ethernet. The network connection rarely fails. However, since Ficus is an experimental file system built into the kernel and undergoing continual change, the machines running it crash or are voluntarily rebooted much more often than most workstations. Machines going up and down effectively create partitions as easily as network failures do.

Two of our workstations are located at project members' homes and are only rarely connected to the network. These primarily disconnected machines store replicas of volumes important to their users. These machines and their volumes communicate with the core Ficus hosts only rarely and only to exchange updates via reconciliation. Although one might expect this pattern of usage to result in very high conflict rates, surprisingly it does not. One reason is that replica reconciliation is scheduled to coordinate the movement of the users with the data of the system, effectively allowing the system's user to act as a human "write token" [7]. While this behavior avoids many conflicts, nevertheless the conflict rate in mostly disconnected volumes is much higher than in other volumes.

Table 1 shows the conflict statistics for more than nine months of operation in the UCLA environment. About 0.0035% of all non-directory updates resulted in conflicts.

During the period under measurement, several conflict resolvers were added to our suite. Using the resolvers available at the end of the measurement period, 162 of the update/update conflicts (roughly, one in three) experienced could have been resolved automatically if the same patterns of conflicts occurred today as did during this nine month period. This set of resolvable conflicts includes files related to the MH mail system, shell history and editor backup files, .newsrsrc

14,142,241 total non-directory updates
14,141,752 non-conflicting updates
489 update/update conflicts
162 automatically resolved
176 resolvable automatically
151 not clearly resolvable automatically
98 update/remove conflicts
98 passed to the user for resolution
708,780 name creations
708,652 non-conflicting name creations
128 name conflicts
128 passed to the user for resolution

Table 1: Conflict statistics for a nine month period. Theoretically resolvable conflicts are conflicts on files with semantics amenable to automatic resolution but for which we have not yet written resolvers.

files, and several types of game score files.

Many of the other conflicts experienced could have been handled by resolvers that have not been written. We found 176 of those, more than another third of the total number of conflicts. These include control files for the trn news reader, saved news postings, manual pages, compiler-produced object files, measurement statistics files, and score files for other games.

The remaining 151 conflicts would not be easy to resolve automatically with our current system. Files that occasionally got into conflict that cannot be resolved include such things as source code and arbitrary text files. A significant number of these conflicts occurred in files placed in orphans. Such files no longer possess their original names. Since most of our existing resolvers are selected solely by name, our current system has little hope of finding a proper resolver for these files. Explicit storage (or identification) of file type would make resolution of these files possible.

None of the name conflicts were resolved automatically. Because many fewer name conflicts occurred than file conflicts, we did not develop any name conflict resolvers in the sample period. About 0.018% of all name creations led to name conflicts, all of which were resolved by human users. On the average, each user had to deal with about ten name conflicts during this nine month period.

Taken as a whole, the average user in this environment thus had to resolve about five conflicts a month, and examine one update/delete conflict per month. In actuality, this average is misleading, since conflicts

tended to happen more often to users who worked with the disconnected machines. A few users thus experienced much higher conflict rates, while many users encountered considerably fewer conflicts than the average.

The conflicts were not evenly spread across all volumes in our environment. Table 2 shows the number of updates and conflicts for different types of volumes, and the conflicts rates by volume for the nine month period.

Volumes are classified as either *shared* or *private*, and either *office*, *disconnected*, or *network*. Shared volumes indicate volumes that receive heavy update traffic from multiple users, often to different volume replicas. A prominent example of this category of volume is the games volume. Updates to the games volume involve access to shared database files (game score files). Multiple users accessing different replicas and using the same application concurrently create high probability of conflicts. In addition, the games volume is a disconnected one, meaning that replicas exist both in the office and at users' homes. Disconnection increases the likelihood of concurrent use, for the time period in which independent updates are deemed concurrent is increased. Fortunately, the shared database files have relatively simple semantics, so it is easy to write automatic resolvers for these files. Other disconnected, shared volumes included volumes of installed programs and libraries, which easily get in conflict if users are not careful about how they perform installations.

The source code volumes are another example of shared volumes, though these are stored entirely in the office. Although most source code files themselves are protected against multiple writers by a revision control service, conflicts can occur in two ways. First, multiple users can attempt to gain write permission on the same file via different replicas. Second, the same user can perform updates to two different replicas. The latter is not nearly as uncommon as it would seem, since source code volumes are replicated on server-style machines, which experience more down-time than normal workstations due to increased load. Server crashes cause automatic replica switching, creating the potential for conflicts: a user updates one replica, then switches replicas and updates the second.

Private volumes are user's personal volumes. They are almost always updated solely by the one user, and experience very few conflicts. However, when the private volumes are also disconnected, conflict rates rapidly increase. Examples of such volumes are the personal volumes of two Ficus project members who have replicas at the office and at home. Home for one user is across town, and home for the other is across the

Volume Classification	Number of Volumes in class	Number of Updates	Number of Conflicts	Conflict Rate	User-Visible Conflict Rate
Disconnected, Shared	9	1,114,855	273	.0245%	.0052%
Disconnected, Private	16	387,523	106	.0274%	.0012%
Office, Shared	27	6,316,331	66	.0010%	.0005%
Office, Private	48	6,286,754	44	.0007%	.0003%
Network, Shared	8	36,778	0	0%	0%

Table 2: Update/update conflicts grouped by volume classification.

ocean in Guam. They both call the office by modem and reconcile periodically, ranging from once a day to once every few weeks.

Most of the private volumes are rarely disconnected, and therefore one would expect there to be almost no conflicts, since only one person is performing updates and usually to the same replica. Private volumes stored only in the office accounted for only 9% of the total conflicts.

Network volumes are volumes shared between sites connected by the Internet. These volumes are all shared, in our current environment. Many of them are test volumes, leading to a low number of updates for such volumes.

As expected, disconnected volumes had a much higher rate of conflicts, 20 to 40 times as high as their office counterparts. Somewhat surprisingly, however, disconnected shared volumes suffer a lower conflict rate than private volumes.

Table 2 also indicates which of these conflicts could have been resolved automatically. For example, all of the conflicts on the game volume were on simple database score files, and therefore easily resolved. The only conflicts that the user need see in the disconnected, shared class of volumes were those in the installation volumes. Most of the conflicts on source code files in the office shared class could not be resolved automatically, however, because source code files have arbitrary semantics, and therefore require user intervention. Those conflicts that were resolvable were on object files (.o files).

The unresolvable conflict rates for disconnected volumes are still significantly higher than the unresolvable conflict rates for office volumes, but the relative difference is somewhat lower, particularly in the case of private volumes. Many of the files in disconnected private volumes that get into conflict are simple database files that can be resolved automatically. Shell histories are again a good example. In the disconnected environment, they are likely to frequently enter conflict, while they rarely enter conflict in the office environment. But these conflicts are always automat-

ically resolvable. Once automatic resolution is taken into account, instead of one disconnected private volume update in five thousand requiring user attention, one in one hundred thousand requires it. This twenty-fold reduction in user intervention in conflicts on this class of volume is a powerful motivation for providing automatic resolution in the disconnected environment.

6 Related Work

The Ficus file system draws from several earlier systems, and has some similarities to work done by others. This section discusses some of the related work, with particular attention to that concerning optimistic replication, conflicts in optimistically replicated systems, and automatic resolution of such conflicts.

Parker's work on version vectors was an important early step in optimistic file replication [14]. It permitted reliable detection of independent updates to different replicas of a data item with limited and reasonable costs for maintaining the necessary information.

Version vectors were used in the university Locus operating system [15, 17], a system that provided data replication and dealt with partitioned operation. However, the Locus system never dealt substantively with the problems of conflicting updates.

Sergio Faissol's Ph.D. dissertation examined this question in the context of databases [2]. He investigated several classes of information that could be stored in a database, how independent updates to those classes of information could be reconciled, and the information required to perform the necessary reconciliation. His work was primarily theoretical, and was never applied to file systems.

The Coda project at Carnegie-Mellon University, discussed in detail in [18], is also developing an optimistically replicated file system. The Coda developers have considered the questions of disconnected operations in a somewhat different context than the Ficus system. They support a highly connected backbone of server machines that replicate files. While these servers may occasionally fail or become disconnected

from each other, they are expected to be more reliable than the typical single-user workstation machine. Client machines cache replicas of files they actually use, and send the updates back to a server replica [10].

The nature of the Coda system makes partitioned first class replicas a less common event than in Ficus. Partitioned update is far more common between first and second class Coda replicas where simpler reconciliation algorithms are possible [10]. References [11, 12] discuss Coda's log-based approach to conflict resolution between first-class replicas. The design of conflict resolution in Coda is described in [13]. Like the Ficus approach, conflict resolvers are provided and are selected by file type.

Unlike Ficus, the Coda approach uses files that hold resolution rules that apply to all files in a directory or its subdirectories. These rules are similar in form to rules in a Unix makefile. By placing a set of generic rules in the topmost directory, Coda can achieve the same effect as Ficus' system resolver file. By using regular expressions that match only certain directory prefixes, the Ficus resolver files can achieve the same effect as Coda's per-directory rules files. Unlike the Ficus approach, Coda does not automatically serially apply different resolvers to a file in conflict, though presumably the makefile rules could be set up in such a way that they could. Generally speaking, the expressive power of the Coda and Ficus approaches seem similar. More experience with both systems is needed to determine if either approach has a clear advantage in user friendliness. The statistics presented in this paper provide the first step at addressing some of these issues.

Huston and Honeyman describe their approach to optimistic replication in disconnected AFS in [9]. This system permits updates to cached copies of data at disconnected client sites under AFS. Writes generated by a disconnected client site are logged and replayed when the client is reconnected to a server. If any of the logged write operations conflict with writes performed by some other client during the disconnection, the conflicts are detected and reported. No attempt is made to automatically resolve them, though Huston and Honeyman do briefly discuss plans to provide tools to help users resolve common types of conflicts.

Howard has developed an optimistic reconciliation-based system to permit occasionally connected machines to share files [8]. He reliably detects conflicts using a journalling mechanism, but currently makes no attempt to reconcile them.

7 Observations and Conclusions

Optimistic file replication in an environment that has any serious degree of disconnection benefits from au-

tomatic conflict resolution; it can substantially reduce the conflict rate observed by users. We present data for two environments, a usually-connected office environment and a periodically connect, usually-disconnected home use environment.

In the office environment, without automatic conflict resolution, the typical user would need to resolve around two conflicts per month, considering both update/update and name conflicts. With automatic resolution, the frequency of conflicts requiring user attention would drop to one and a half or less. The resolvers Ficus currently has installed and that will be added to our suite can reduce the total number of user-visible conflicts by about one half.

The effects are more dramatic in the home use environment. In this environment, two users generated 380 conflicts in 9 months, averaging nearly a conflict a day for each user. In actuality, one of the two users experienced the bulk of the conflicts. He made extensive use of disconnected home computing, reconciling his volumes only once a day or so, so his conflict rate was significantly higher. He observed 30 to 40 conflicts per month. Applying automatic resolvers to the home use environment reduces the observed conflict rate for this user to around seven conflicts per month.

The in-office statistics might suggest that the added value of automatic resolution of some conflicts is not that great, in that environment. However, there are some additional points to consider. First, as pointed out in Section 5, we did not gather statistics for the value of the most important case of all, the automatic, built-in directory resolver. (We hope to gather these statistics in the future.) Second, many of the conflicts that are automatically resolved are easily handled using a program, but hard to resolve by hand. If they were not automatically resolved, they would require a user to invoke a tool that might equally well be invoked automatically. Directories and binary data are examples. Third, further effort applied to writing resolvers certainly would decrease the observed rate of conflicts even more.

The case for automatic conflict resolution in less connected environments is even stronger. Environments in which disconnection is more even common than our home use environment, such as mobile computing, can be expected to have higher conflict rates. Our data suggests that conflicts in this environment are often easier to reconcile than those in the office environment. Decreasing the observed conflict rate by sixfold for a replicated home use environment is a major improvement.

Many files have semantics allowing fairly simple resolution of all conflicts. Even when not all possible conflicts a file can experience are automatically resolv-

able, there are often large classes of conflicts that can be fixed without human attention. Unix-style directories are one such example, where all conflicts except name conflicts can be automatically resolved. In several other cases, we have discovered that solutions that solve 80% or so of all possible problems work very well. The user need only be informed in the case of the 20% that cannot be resolved. In some cases, the resolution of the difficult set of conflicts can even be guessed at, with the user only becoming aware of the difficulty if the guess is wrong. `.newsrc` and conflict log files are two such cases.

Implementing data storage as directories offers an opportunity to leverage the Ficus directory resolution algorithms. When data follows insert/delete semantics (such as Ficus graft points do) this mapping is quite natural. In the future, we plan to restructure the directory reconciliation algorithms as a library that can be used in more general situations.

Reconciliation chooses resolvers first by file name, applying consecutive resolvers until one succeeds. Storing the file type as an attribute of the file would be a more attractive approach. Existing Unix file attributes leave little room for such information, but an object-oriented file system with a general purpose attribute service could store a resolver list as an attribute. The reconciliation process could then directly call the proper resolvers for each conflict. Such an object-oriented file system is under development in our project, and will be tested with resolver attributes. Until all data is stored as typed objects, the approach discussed in this paper offers an attractive interim solution.

While this work has been applied to a Unix-style file system, most of it is not specific to Unix systems. The general approach is applicable to many other systems and could be simplified on systems that don't allow multiple names for the same file. The approach of pairwise resolution of single conflicting files, name-based choice of resolvers, and iteratively invoking conflict resolvers until one of them succeeds appears to be generally applicable.

In conclusion, our experience with conflicts in optimistically replicated file systems is that, for one common environment, conflicts are rare. At least two thirds of those conflicts that do occur can be resolved automatically, with no user intervention or even notification. Further effort in building more resolvers would reduce the rate of user notification of conflicts even lower. Our experience with working on the Ficus system is that the typical user is not bothered by either the possibility of conflicts or their actual occurrence.

Acknowledgments

Ficus is a large system and would not be possible without the efforts of many people. In addition to the authors of this paper, Richard Guy, Dieter Rothmeier, and Steven Stovall were involved in the issues discussed in this paper. Others who have contributed to Ficus include (chronologically) Wai Mak, Tom Page, Yuguang Wu, Jeff Weidner, John Salomone, Michial Gunter, Ashvin Goel, Geoff Kuenning, Sivaprakasam Suresh, Sugata Mukhopadhyay, and Ted Kim.

References

- [1] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3):341–370, September 1985.
- [2] Sergio Zarur Faissol. *Operation of Distributed Database Systems Under Network Partition*. Ph.D. dissertation, University of California, Los Angeles, 1981.
- [3] Richard G. Guy. *Ficus: A Very Large Scale Reliable Distributed File System*. Ph.D. dissertation, University of California, Los Angeles, June 1991. Also available as UCLA technical report CSD-910018.
- [4] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek, and Dieter Rothmeier. Implementation of the Ficus replicated file system. In *USENIX Conference Proceedings*, pages 63–71. USENIX, June 1990.
- [5] Richard G. Guy and Gerald J. Popek. Reconciling partially replicated name spaces. Technical Report CSD-900010, University of California, Los Angeles, April 1990.
- [6] Richard G. Guy, Gerald J. Popek, and Thomas W. Page, Jr. Consistency algorithms for optimistic replication. In *Proceedings of the First International Conference on Network Protocols*. IEEE, October 1993.
- [7] John S. Heidemann, Thomas W. Page, Jr., Richard G. Guy, and Gerald J. Popek. Primarily disconnected operation: Experiences with Ficus. In *Proceedings of the Second Workshop on Management of Replicated Data*. IEEE, November 1992.
- [8] John H. Howard. Using reconciliation to share files between occasionally connected computers. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 56–60, Napa, California, October 1993. IEEE.
- [9] L. B. Huston and Peter Honeyman. Disconnected operation for AFS. In *Proceedings of the USENIX Symposium on Mobile and Location-Independent Computing*, pages 1–10. USENIX, 1993.
- [10] James J. Kistler and Mahadev Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, 1992.

- [11] Puneet Kumar. Coping with conflicts in an optimistically replicated file system. In *Proceedings of the Workshop on Management of Replicated Data*, pages 60–64. IEEE, November 1990.
- [12] Puneet Kumar and Mahadev Satyanarayanan. Log-based directory resolution in the Coda file system. Technical Report CMU-CS-91-164, Carnegie-Mellon University School of Computer Science, 1991.
- [13] Puneet Kumar and Mahadev Satyanarayanan. Supporting application-specific resolution in an optimistically replicated file system. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 66–70, Napa, California, October 1993. IEEE.
- [14] D. Stott Parker, Jr., Gerald Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, 9(3):240–247, May 1983.
- [15] Gerald Popek, Bruce Walker, Johanna Chow, David Edwards, Charles Kline, Gerald Rudisin, and Greg Thiel. LOCUS: A network transparent, high reliability distributed system. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pages 169–177. ACM, December 1981.
- [16] Gerald J. Popek, Richard G. Guy, Thomas W. Page, Jr., and John S. Heidemann. Replication in Ficus distributed file systems. In *Proceedings of the Workshop on Management of Replicated Data*, pages 20–25. IEEE, November 1990.
- [17] Gerald J. Popek and Bruce J. Walker. *The Locus Distributed System Architecture*. The MIT Press, 1985.
- [18] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.

Author Information

Electronic mail to the authors should be directed to ficus@ficus.cs.ucla.edu.

Peter Reiher received his B.S. in Electrical Engineering from the University of Notre Dame in 1979. He received his M.S. in Computer Science from UCLA in 1984, and his Ph.D. in Computer Science in 1987. He has worked on several distributed operating systems projects. His research interests include distributed operating systems, optimistic computation, and security for distributed systems.

John Heidemann received his B.S. from the University of Nebraska-Lincoln and his M.S. from UCLA, both in computer science. He is currently pursuing his Ph.D. in stackable layered file systems at UCLA.

David Ratner has been a graduate student researcher with the Ficus project at UCLA for three years. His work on the project includes the rewriting and maintaining of the reconciliation, directory and file management code. Recently he has been working on algorithm and replication issues. He received his B.A. in Computer Science and a B.A. in Mathematics from Cornell University in 1991.

Gerald J. Popek has been a Professor of Computer Science at UCLA since 1973. He has been the principal investigator for the ARPA distributed systems contract since 1977. He is best known for his work on secure systems, the design of the Locus distributed Unix system, and most recently, the Ficus large scale replicated filing environment.

Popek's academic background includes a doctorate in computer science from Harvard University. He co-authored "The LOCUS Distributed System Architecture," published by the MIT Press in 1985, and has written more than 70 professional articles concerned with computer security, system software, and computer architectures.

He is a principal founder of Locus Computing Corporation. Privately-held Locus Computing is the largest independent developer of UNIX-based connectivity and distributed processing software technology.

Greg Skinner is a systems programmer for the Ficus Distributed Systems Research Group of the UCLA Computer Science Department. He received his MSCS from UCLA in 1992, with a concentration in network modeling and analysis. In his ten year professional career, he has worked on a number of projects in the areas of distributed systems and computer networks.

Reducing File System Latency using a Predictive Approach *

James Griffioen, Randy Appleton

Department of Computer Science

University of Kentucky

Lexington, KY 40506

Abstract

Despite impressive advances in file system throughput resulting from technologies such as high-bandwidth networks and disk arrays, file system latency has not improved and in many cases has become worse. Consequently, file system I/O remains one of the major bottlenecks to operating system performance [10].

This paper investigates an automated predictive approach towards reducing file latency. *Automatic Prefetching* uses past file accesses to predict future file system requests. The objective is to provide data in advance of the request for the data, effectively masking access latencies. We have designed and implemented a system to measure the performance benefits of automatic prefetching. Our current results, obtained from a trace-driven simulation, show that prefetching results in as much as a 280% improvement over LRU especially for smaller caches. Alternatively, prefetching can reduce cache size by up to 50%.

1 Motivation

Rapid improvements in processor and memory speeds have created a situation in which I/O, in particular file system I/O, has become the major bottleneck to operating system performance [10]. Recent advances in high bandwidth devices (e.g., RAID, ATM networks) have had a large impact on file system throughput. Unfortunately, access latency still remains a problem and is not likely to improve significantly due to the physical limitations of storage devices and network transfer latencies. Moreover, the increasing popularity of certain file system designs such as RAID, CDROM, wide area distributed file systems, wireless networks, and mobile hosts has only exacerbated the latency problem. For example, distributed file systems experience network latency combined with standard disk latency. As

distributed file systems scale both numerically and geographically, as envisioned by the Andrew File System designers [7], network delays will become the dominant factor in remote file system access. Similarly, local file systems built on technologies like CD-ROMs also suffer from very high latencies but continue to increase in popularity due to the large amount of storage space they offer.

Although a variety of high bandwidth technologies are now available, it is unlikely that existing (and emerging) low-end technologies such as serial lines running SLIP or PPP, 64/128 Kb ISDN and other slower speed networks will disappear in the near future given their low-cost and wide-spread use. Such communication technologies suffer from both high latencies and low bandwidths. Distributed file systems that build on or incorporate these technologies will experience latencies substantially higher than that of conventional file systems. However, the appeal of low-cost widely available shared access to files will certainly prolong the existence of such file systems, despite their poor performance.

The goal of our research is to investigate methods for successfully reducing the perceived latency associated with file system operations. In this paper, we describe a new method for masking file system latency called *automatic prefetching*. Automatic prefetching takes a heuristic-based approach using knowledge of past accesses to predict future access without user or application intervention. As a result, applications automatically receive reduced perceived latencies, better use of available bandwidth via batched file system requests, and improved cache utilization.

2 Related work

Both caching and prefetching have been used in a variety of settings to improve performance. The following briefly describes related work involving caching and prefetching to improve file system performance.

*This work was supported in part by NSF grant number CCR-9309176

2.1 Caching

Caching has been used successfully in many systems to substantially reduce the amount of file system I/O [16, 6, 8, 1]. Despite the success of caching, it is precisely the accesses that cannot be satisfied from the cache that are the current bottleneck to file system performance [10]. Unfortunately, increasing the cache size beyond a certain point only results in minor performance improvements. Experience shows that the relative benefit of caching decreases as cache size (and thus cache cost) increases [9, 8]. There exists a threshold beyond which performance improvements are minor and prohibitively expensive. Moreover, studies show that the “natural” cache size or threshold is becoming a substantially larger fraction (one forth to one third) of the total memory, due in part to larger files (e.g., big applications, databases, video, audio, etc.) [2]. Consequently, new methods are needed to reduce the perceived latency of file accesses and keep cache sizes in check.

Although machines with large memories are now available, low-end workstations, PCs, mobile laptops/notebooks, and now PDAs (personal data assistants) with limited memory capacities enjoy widespread use. Because of cost or space constraints these machines cannot support large file caches. The desire for smaller portable machines combined with continually increasing files size means that large caches cannot be assumed to be the complete solution to the latency problem.

Finally, as a result of rapid improvements in bandwidth, cache miss service times are dominated by latency. Note that:

- Most files are quite small. In fact, measurements of existing distributed file systems show that the average file is only a few kilobytes long [9, 2]. For files of this size, transmission rate is of little concern when compared to the access latency across a WAN or from a slow device. As a result, access latency, not bandwidth, becomes the dominate cost for references to files not in the cache.
- In many distributed file systems, the open() and close() functions represent synchronization points for shared files. Although the file itself may reside in the client cache, each open() and close() call must be executed at the server for consistency reasons. The latency of these calls can be quite large, and tends to dominate other costs, even when the file is in the file cache.

In short, the benefits of standard caching have been realized. To improve file system performance further

and keep file cache sizes in check, caching will need to be supplemented with new methods and algorithms.

2.2 Prefetching

The concept of prefetching has been used in a variety of environments including microprocessor designs, virtual memory paging, databases, and file read ahead. More recently, long term prefetching has been used in file systems to support disconnected operation [14, 15, 5]. Prefetching has also been used to improve parallel file access on MIMD architectures [4].

One relatively straight forward method of prefetching is to have each application inform the operating system of its future requirements. This approach has been proposed by Patterson et. al. [11]. Using this approach, the application program informs the operating system of its future file requirements, and the operating system then attempts to optimize those accesses. The basic idea is that the application knows what files will be needed and when they will be needed.

Application directed prefetching is certainly a step in the right direction. However, there are several drawbacks to this approach. Using this approach, applications must be rewritten to inform the operating system of future file requirements. Moreover, the programmer must learn a reasonably complex set of additional system directives that must be strategically deployed throughout the program. This implies that the application writer must have a thorough understanding of the application and its file access patterns. Ironically, a key goal of many recent languages, in particular object-oriented languages, is abstraction and encapsulation; hiding the implementation details from the programmer. Even when the details are visible, our experience indicates that the enormity and complexity of many software systems creates a situation in which experts may have difficulty grasping the complete picture of file access patterns. Moreover, incorrectly placed directives or an incomplete set of directives can actually degrade performance rather than improve it.

A second problem is that the operating system needs a significant lead-time to insure the file is available when needed. Therefore, in order to benefit from prefetching, the application must have a significant amount of computation to do between the time the file is predicted and the time the file is accessed. However, many applications do not know which files they will need until the actual need arises. For instance, the pre-processor of a compiler does not know the pattern of nested include files until the files are actually encountered in the input stream, nor will an editor necessarily know which files a user normally edits. Our approach attempts to solve this problem by predicting the need

for a file well in advance of when the application could; in some cases long before the application even begins to execute.

A third problem with application driven prefetching arises in situations where related file accesses span multiple executables. Typically applications are written independently and only know file access patterns within the application. In situations where a series of applications execute repeatedly, like an edit/compile/run cycle, or certain commonly run shell scripts, no one application knows the cross-application file access patterns, and therefore cannot inform the operating system of a future application's file requirements. In some cases, batch-type utilities, such as the Unix make facility, can be instrumented to understand cross-application access patterns. However, even in this case, a complete view of the real cross application pattern is often unknown to the user or requires extreme expertise to determine the pattern. Our approach uses long term history information to support prefetching across application boundaries.

3 Automatic Prefetching

We are investigating an approach we call *automatic prefetching*, in which the operating system rather than the application predicts future file requirements. The basic idea and hypothesis underlying automatic prefetching is that future file activity can be successfully predicted from past file activity. This knowledge can then be used to improve overall file system performance.

Automatic prefetching has several advantages over existing approaches. First, existing applications do not need to be rewritten or modified, nor do new applications need to incorporate non-portable prefetching operations. As a result, all applications receive the benefits of automatic prefetching, including existing software. Second, because the operating system automatically performs prefetching on the application's behalf, application writers can concentrate on solving the problem at hand rather than worrying about optimizing file system performance. Third, the operating system monitors file access across application boundaries and can thus detect access patterns that span multiple applications executed repeatedly. Consequently, the operating system can prefetch files substantially earlier than the file is actually needed, often before the application even begins to execute.

Automatic prefetching allows the operating system effectively to overlap processing with file transfers. The operating system can also use past access information to batch together multiple file requests and thus make better use of available bandwidth. Past access in-

formation can also be used to improve the cache management algorithm, effectively reducing cache misses even if no prefetching occurs.

The first goal of our research was to determine whether such an approach is viable. Our second goal was to develop effective prefetch policies and quantify the benefits of automatic prefetching. The following sections consider each of these objectives and describe our results.

4 Analysis of Existing Systems

To determine the viability of automatic prefetching, we analyzed current file system usage patterns. Although other researchers have gathered file system traces [9, 2], we decided to modify the SunOS kernel in order to gather our own traces that extract specific information important to our research. In addition to recording all file system calls made by the system, the kernel gathers precise information regarding the issuing process and the timing for every operation. The timing information not only serves as an indicator of the system's performance, but it also provides information as to whether prefetching can have any substantial effects on performance.

We gathered a variety of traces, including the normal daily usage of several researchers, and also various synthetic workloads. Traces were collected on a single Sun Sparcstation supporting several users executing a variety of tasks. Traces were collected for varying time periods with the longest traces spanning more than 10 days and containing over 500,000 operations. Users were not restricted in any way. Typical daily usage included users processing email, editing, compiling, preparing documents and executing other task typical of an academic environment. This particular set of traces contains almost no database activity. The data we collected appears to be in line with that of other studies [9, 2] given similar workloads.

Our initial analysis of the trace data indicates that typical file system usage can realize substantial performance improvements from the use of prefetching, and also provides several guidelines for a successful prefetching policy.

First, the data shows that there is relatively little time between the moment when a file is opened and the moment when the first read occurs (see figure 1). In fact, the median time for our traces was less than three milliseconds. Consequently, prefetching must occur significantly earlier than the open operation to achieve any significant performance improvement. Prefetching at open time will only provide minor improvements.

Second, the data shows that the average amount of time between successive opens is substantial (200

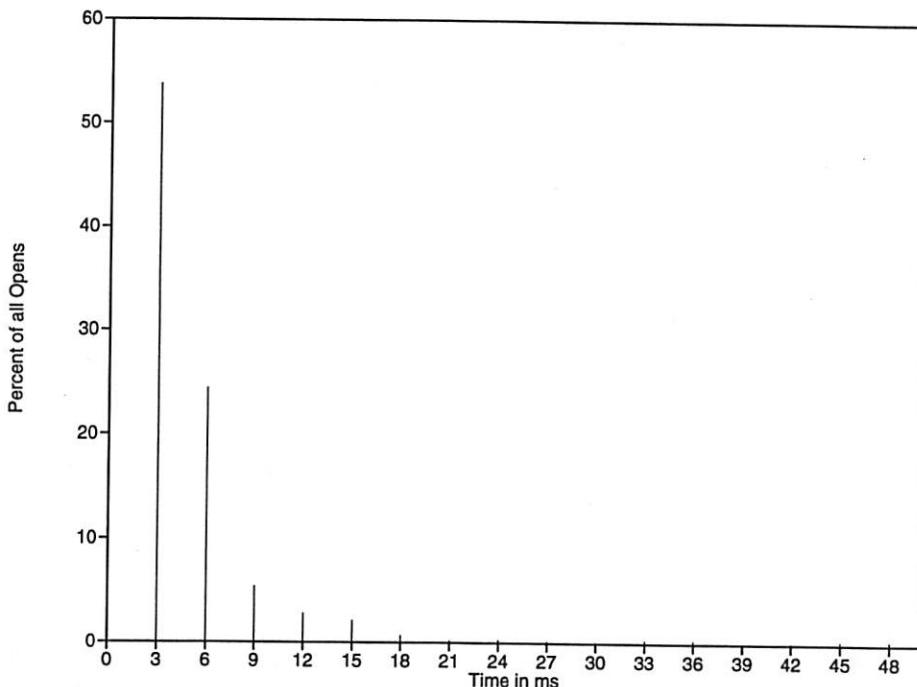


Figure 1: Histogram of times between open and first read of a file.

ms). If the operating system can accurately predict the next file that will be accessed, there exists a sufficient amount of time to prefetch the file.

In a multi-user, multiprogramming environment, concurrently executing tasks may generate an interleaved stream of file requests. In such an environment, reliable access patterns may be difficult to obtain. Even when patterns are discernable, the randomness of the concurrency may render the prefetching effort ineffective. However, analysis of trace data consisting of multiple users (and various daemons) shows that even in a multiprogramming environment accesses tend to be “sequential” where we define sequential as a sensible/predictable uninterrupted progression of file accesses associated with a task. In fact, measurements show that over 94% of the accesses follow logically from the previous access. Thus multiprogramming seems to have little effect on the ability to predict the next file referenced.

5 The Probability Graph

We have designed and implemented a simple analyzer that attempts to predict future accesses based on past access patterns. Driven by trace data, the analyzer dynamically creates a logical graph called a *Probability Graph*. Each node in the graph represents a file in the file system.

Before describing the probability graph, we must de-

fine the *lookahead period* used to construct the graph. The lookahead period defines what it means for one file to be opened “soon” after another file. The analyzer defines the lookahead period to be a fixed number of file open operations that occur after the current open. If a file is opened during this period, the open is considered to have occurred “soon” after the current open. A physical time measure rather than a virtual time measure could be used, but the above measure is easily obtained and can be argued to be a better definition of “soon” given the unknown execution times and file access patterns of applications. Our results show that this measure works well in practice.

We say two files are *related* if the files are opened within a lookahead period of one another. For example, if the lookahead period is one, then the next file opened is the only file considered to be related to the current file. If the lookahead period is five, then any file opened within five files of the current file is considered to be related to the current file.

The analyzer allocates a node in the probability graph for each file of interest in the file system. Unix exec system calls are treated like opens and thus are included in the probability graph. One graph, derived from the trace described in section 7, generated approximately 6,500 nodes accessed over an eight day period. Each node consumes less than one hundred bytes, and can be efficiently stored on disk in the inode of each associated file, with active portions cached for

better performance. Our current graph storage scheme has not been optimized and thus is rather wasteful. We have recently begun investigating methods that will substantially reduce the graph size via graph pruning, aging, and/or compression.

Arcs in the probability graph represent related accesses. If the open for one file follows within the lookahead period of the open for a second file, a directed arc is drawn from the first to the second. Larger lookaheads produce more arcs. The analyzer weighs each arc by the number of times that the second file is accessed after the first file. Thus, the graph represents an ordered list of files demanded from the file system, and each arc represents the probability of a particular file being opened soon after another file.

Figure 2 illustrates the structure of an example probability graph. The probability graph provides the in-

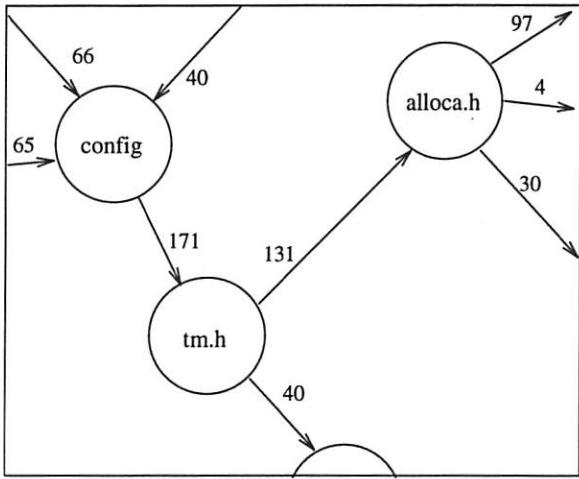


Figure 2: Three nodes of an example probability graph.

formation necessary to make intelligent prefetch decisions. We define the *chance* of a prediction being correct as the probability of a file (say file B) being opened given the fact that another file (file A) has been opened. The chance of file B following file A can be obtained from the probability graph as the ratio of the number of arcs from file A to file B divided by the total number of arcs leaving file A. We say a prediction is *reasonable* if the estimated chance of the prediction is above a tunable parameter *minimum chance*. We say a prediction is *correct* if the file predicted is actually opened within the lookahead period.

Establishing a minimum chance requirement is crucial to avoid wasting system resources. In the absence of a minimum requirement, the analyzer would produce several predictions for each file open, consuming network and cache resources with each prediction, many of which would be incorrect.

To measure the success of the analyzer we define an *accuracy* value. The accuracy of a set of predictions is the number of correct predictions divided by the total number of predictions made. The accuracy will almost always be at least as large as the minimum chance, and in practice is substantially higher.

The number of predictions made per open call varies with the required accuracy of the predictions. Requiring very accurate predictions (predictions that are almost never wrong) means that only a limited number of predictions can be made. For one set of trace data, using a relatively low minimum chance value (65%) the predictor averaged 0.45 files predicted per open. For higher minimum chance values (95%) the predictor averaged only 0.1 files predicted per open. Even when using a relatively low minimum chance (e.g., 65%), the predictor was able to make a prediction about 40% of the time and was correct on approximately 80% of the predictions made.

Figure 3 shows the distribution of estimated chance values with a lookahead of one. The distribution shows that a large number of predictions have an estimated chance of 100%. Setting the minimum chance less than 50% places the system in danger of prefetching many unlikely files. By setting the minimum chance at 50%, very few files that should have been prefetched will be missed. Moreover, the distribution shows how a low minimum chance can still result in a high average accuracy.

6 A Simulation System

To evaluate the performance of systems based on automatic prefetching, we implemented a simulator that models a file system. In order to simulate a variety of file system architectures having a variety of performance characteristics, the simulator is highly parameterized and can be adjusted to model several file system designs. This flexibility allows us to measure and compare the performance of various cache management policies and mechanisms under a wide variety of file system conditions. The simulator consists of four basic components: a *driver*, *cache manager*, *disk subsystem*, and *predictor*.

The *driver* reads a timestamped file system trace and translates each file access into a file system request for the simulator to process. Because the driver generates file requests directly from the trace data, the workload is exactly like that of typical (concurrent) user-level applications. However, the driver must modify the set of requests in a few special cases. Because the simulator is only interested in file system I/O activity, the driver removes accesses made to files representing devices such as terminals or /dev/null. References to

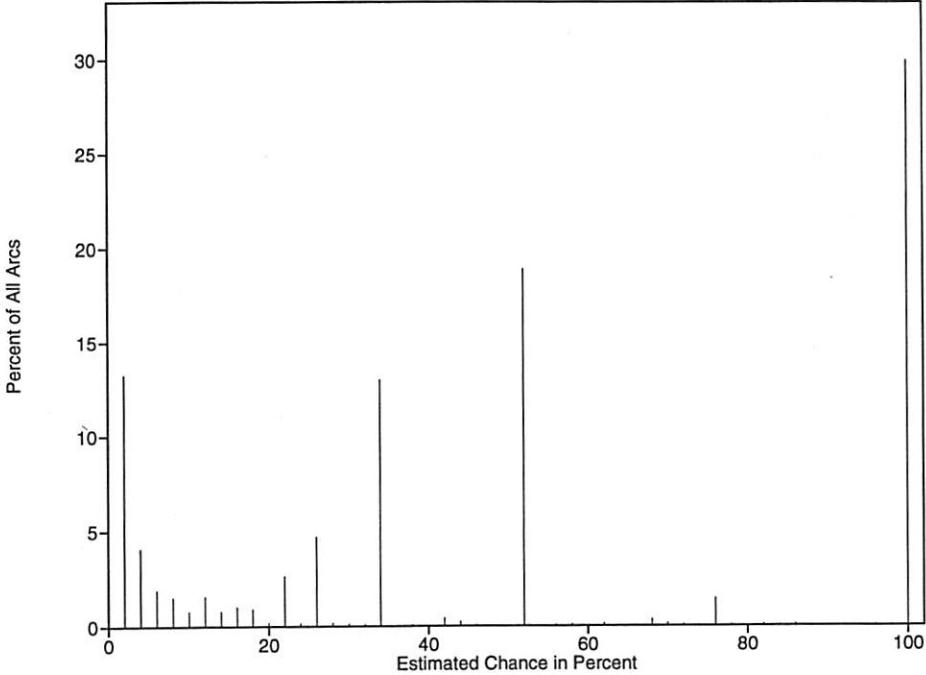


Figure 3: Histogram of estimated chances given a lookahead of one.

certain standard shared libraries such as the C library are also eliminated. Accesses (e.g., `mmap()` calls) to these libraries rarely require any file system activity, since they are typically already present in the virtual memory cache.

The *cache manager* manages a simulated file cache and services as many requests as possible from the cache without invoking the disk subsystem. We have implemented two cache managers. The first is a standard *LRU cache manager*, where disk pages are replaced in the order of least recent use. The second cache manager is the *prefetch cache manager*. The prefetch cache manager operates much like the LRU manager, updating timestamps on each access and replacing the least recently used page. However, the prefetch manager also updates timestamps based on knowledge of expected accesses from the predictor, thus rescuing some-soon-to-be-accessed pages from replacement. We have found that prefetch cache management can improve performance even if no prefetching occurs (i.e., no pages are actually brought in ahead of time). When run in prefetch mode, the simulator shows that anywhere between 5% and 30% of the performance improvement comes from pages that were rescued rather than actually being prefetched.

The task of the *disk subsystem* is to simulate a file storage device. The current disk subsystem has been configured to emulate local disks. Local disk have relatively low latency when compared to our other target

file systems (e.g., wide area distributed file systems, CDROMs, RAIDs, or wireless networks). Consequently, we expect that the performance improvements realized with a local disk model will only be amplified in our other target environments. In the following tests, we assumed a disk model with a first access latency of 15 ms and a transfer rate of 2 MB/sec after factoring in typical file system overhead.

Finally, the simulator contains a *predictor*. The predictor observes open requests that arrive from the driver, and records the data in the probability graph described earlier. The predictor builds the probability graph dynamically just as it would be done in a real system. The longer the simulator executes, the wiser it becomes. On each access the simulator gains a clearer understanding of the true access patterns.

During each open, the probability graph is examined for prefetch opportunities. If an opportunity is discovered, then a read request is sent to the cache manager. If the cache contains the appropriate data, then the data's access time is set to the current time. This ensures that the data will be present for the anticipated need, and possibly rescues the data from an impending flush from the cache. If the prefetch request cannot be satisfied from the cache, then it is prefetched from the disk subject to the characteristics of the disk subsystem.

Notice that the current disk subsystem does no reordering of requests. In particular, it does not preempt or defer prefetch requests to satisfy subsequent appli-

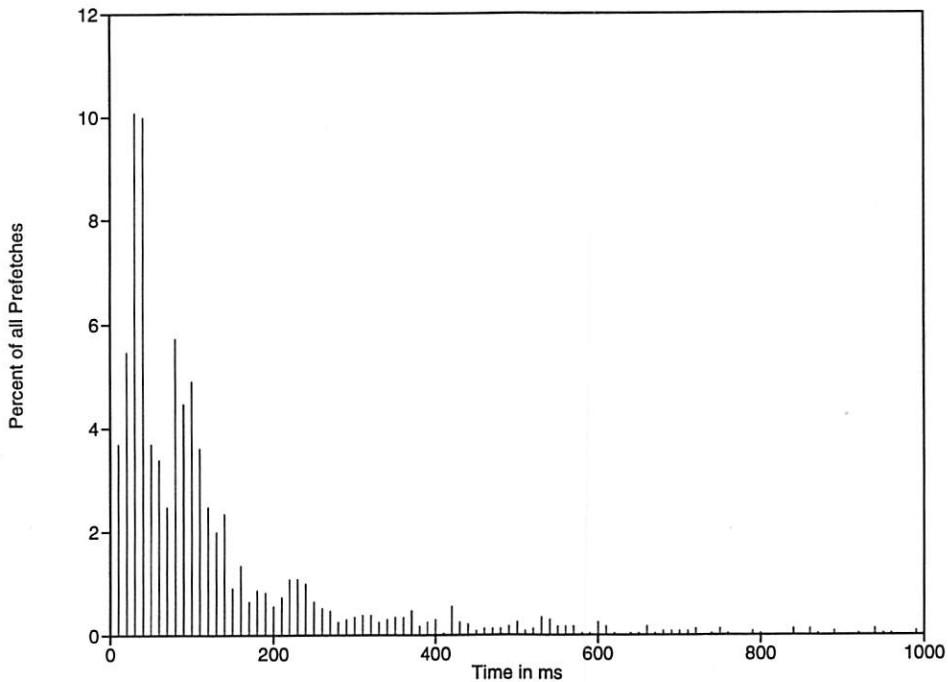


Figure 4: Histogram of times between prefetch and first read access.

cation requests. Reordering and prioritizing requests represents an area of further potential performance improvements.

We are currently in the process of implementing the automatic prefetching system inside a Unix kernel running NFS to measure performance on an actual system.

7 Experimental Results

We performed several tests to measure the performance improvements achieved by automatic prefetching. For the particular set of tests described below, a trace taken over an eight day period containing the unrestricted activity of multiple users was used. To determine the performance benefits of prefetching, we ran several simulations varying the cache size, lookahead value, and minimum chance and also measured the LRU performance in each case for comparison purposes.

Recall from section 4, that the time between the open of a file and the first read is too small for prefetching to be effective. Figure 4 shows that the simulator is able to predict and begin prefetching files sufficiently far in advance of the first read to the file. Our measurements indicate that 94% of the files that were predicted and then subsequently accessed were prefetched more than 20 ms before the actual need, resulting in cache hits at the time of the first read.

7.1 Prefetch Parameters Effect on Performance

Two parameters that significantly affect the predictions made by the predictor are the *lookahead* and *minimum chance* values.

Recall that the lookahead represents how close two file opens need be for the files to be considered related. Setting this value very large increases the number of files that are considered related to each other, and therefore each file open may potentially cause several other files to be prefetched.

Large lookaheads increase the number of files prefetched since more predictions are made in response to each open request. Moreover, large lookaheads result in files being prefetched substantially earlier, because predictions can be made much further in advance. As a result, large lookaheads are inappropriate for smaller cache sizes, but often perform very well with larger caches¹. In the case of small caches, large lookaheads tend to prefetch files too far in advance of the need. As a result, data necessary to the current computation may be forced out of the cache and replaced

¹Here we use the terms "small" and "large" as relative measures of cache size where the meaning of "small" and "large" depend on the workload. A "small cache" will have many cache misses while a "large cache" will have few misses. For the workload in this trace, caches of one megabyte or less would be considered small while caches of three megabytes or more would be considered large. Other traces would produce different values.

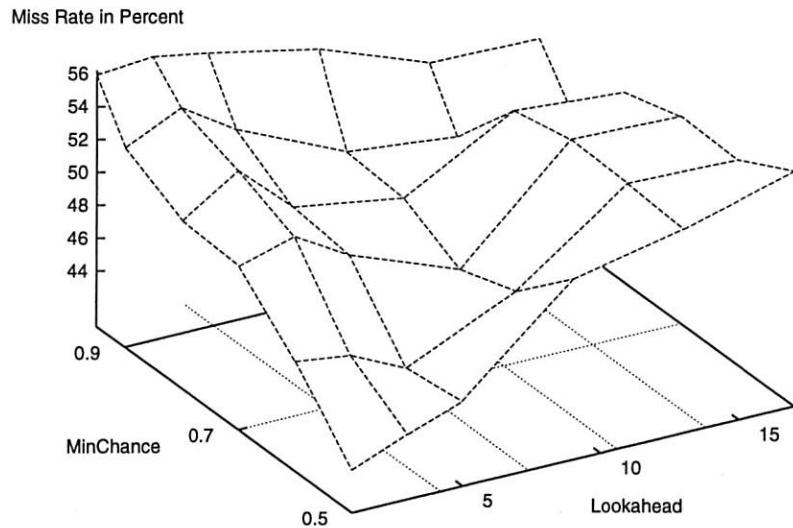


Figure 5: Cache misses as function of lookahead and MinChance for a 400K cache. Performance varies by as much as 13% (between 43% and 56%) depending on the lookahead and minchance settings.

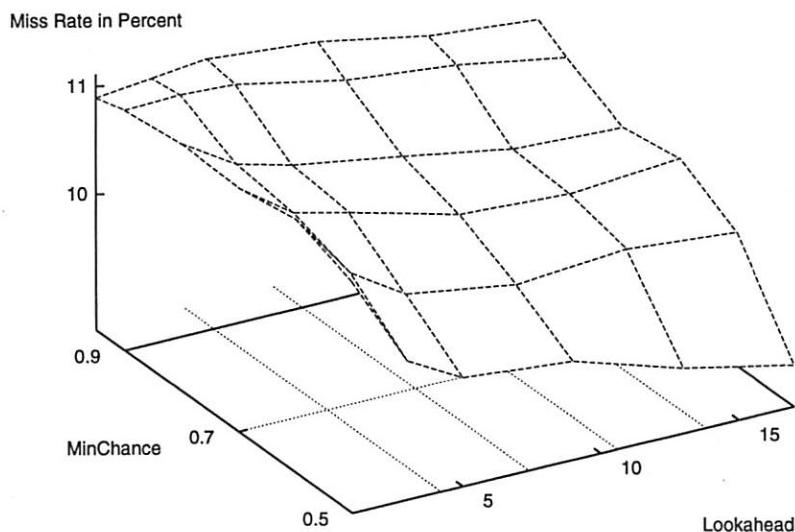


Figure 6: Cache misses as function of lookahead and MinChance for a 4M cache. Performance varies by as much as 2% (between 9% and 11%) depending on the lookahead and minchance settings.

400K Cache Miss Rates (%)							
		Lookahead					
		1	3	5	9	13	17
MinChance	50%	43.2	44.5	45.7	51.6	53.0	54.9
	60%	47.3	46.8	45.2	48.3	53.3	53.1
	70%	50.6	51.6	49.6	47.1	53.5	53.2
	80%	50.8	53.0	50.0	48.9	52.7	52.2
	90%	52.8	54.3	52.2	49.2	48.5	48.9
	95%	55.9	56.2	55.6	54.2	51.8	51.6

Table 1: Data points corresponding to Figure 5.

4000K Cache Miss Rates (%)							
		Lookahead					
		1	3	5	9	13	17
MinChance	50%	10.9	10.0	9.8	9.7	9.4	9.1
	60%	11.1	10.5	10.2	10.0	10.1	9.9
	70%	11.0	10.6	10.5	10.2	10.2	10.2
	80%	11.0	10.7	10.6	10.4	10.2	10.1
	90%	11.0	11.0	10.9	10.7	10.6	10.4
	95%	10.9	10.9	11.0	10.9	10.7	10.6

Table 2: Data points corresponding to Figure 6.

by (useless) data needed far in the future. However, for larger cache sizes, the cache may have sufficient space to load in file data required in the future without disturbing the file data required by the current computation.

MinChance is the minimum estimated probability that a given file will be needed in the near future. For larger cache sizes smaller MinChance values perform better. Setting the MinChance low results in aggressive prefetching. When the cache is large, incorrect prefetches have minimal affect on overall performance. Somewhat surprisingly, an aggressively low MinChance value benefits small caches as well. Because the hit rate is low for small caches, correct predictions result in large performance benefits. A low minimum chance increases the total number of correct predictions. For moderate cache sizes, the optimal MinChance is a function of the specific cache size and must limit the number of missed prefetch opportunities without prefetching unnecessary files.

In summary, MinChance should be low (aggressive) for both large and small caches, but higher for intermediate size caches. Lookahead should increase with increasing cache size. Figures 5 and 6 and their associated tables, tables 1 and 2, illustrate these tradeoffs

for a 400 KB cache and a 4000 KB cache respectively. Clearly, the Lookahead and MinChance parameters are highly sensitive to the cache size and must be adjusted in accordance with the cache size. Moreover, multiple settings for a particular cache size may result in approximately equal miss ratios. In this case, other factors such as network congestion and processing overhead can be used to aid in the selection of appropriate parameter settings.

7.2 Performance Compared to LRU

The primary goal of automatic prefetching is to bring necessary file data into the cache before it is needed. If automatic prefetching is successful we would expect the number of cache misses to be less than the number of cache misses experienced under standard LRU cache management.

Figure 7 shows the number of page misses that the file system incurred under LRU and under prefetching for various cache sizes. After tuning the above parameters, prefetching performs better than LRU for all cache sizes, in some cases outperforming LRU by as much as 280%. Also note that for the cache sizes shown here, prefetching provided the same or better performance

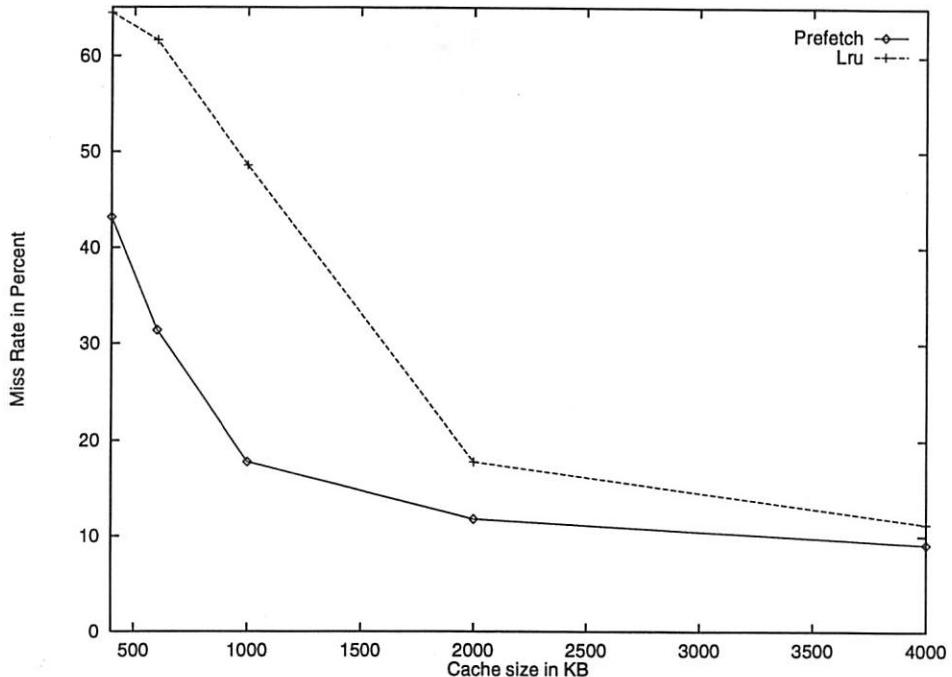


Figure 7: Cache misses as a function of cache size.

than LRU using a cache half the size. This is particularly important for machines that do not have large amounts of memory available for file caching. Even for large memory machines, the ability to achieve similar performance using smaller cache sizes results in more memory for applications. This also indicates that the number of correctly prefetched pages more than offsets any pages incorrectly forced out of the cache by prefetching, even for small cache sizes.

For this particular trace, both LRU and prefetching realize relatively little improvement in the miss ratios for caches larger than 4 MB². However, although LRU performance begins to approach prefetch performance as cache size increases, simulations out to cache sizes of 20 MB still show that prefetching results in an 11% reduction in the number of misses as compared to LRU.

8 Conclusions

Our results show that reasonable predictions can be made based on past file activity. As a result, automatic prefetching can substantially reduce I/O latency, make better use of the available bandwidth via batched prefetch requests, and improve cache utilization. As wide area distributed file systems, CDROM, RAID,

and other high latency/high bandwidth systems become prevalent, prefetching will become an increasingly important mechanism toward high-performance I/O.

9 Acknowledgements

We would like to thank the reviewers for their helpful comments and suggestions. We would also like to thank Mary Baker for reviewing an early draft of the paper and providing valuable feedback. Finally we would like to thank the DCS users for submitting to being traced.

References

- [1] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-Volatile Memory for Fast, Reliable File Systems. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 10–22, October 1992.
- [2] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 198–212. Association for Computing Machinery SIGOPS, October 1991.

²Like the traces reported in [2], this particular trace consisted of unrestricted real user usage. However, unlike the traces in [2], this trace contained no “heavy users” and thus can achieve reasonable miss rates with a 4 MB cache.

- [3] James Griffioen and Randy Appleton. Automatic Prefetching in a WAN. In *Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 8 – 12, Oct 1993.
- [4] D. Kotz and C. Ellis. Prefetching in file systems for MIMD multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1:218–230, 1990.
- [5] Geoff Kuenning, Gerald J. Popek, and Peter Reiner. An Analysis of Trace Data for Predictive File Caching in Mobile Computing. In *Proceedings of the 1994 Summer USENIX Conference*, June 1994.
- [6] Samuel J. Leffler, Marshal K. Mc Kusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3 BSD Unix Operating System*. Addison Wesley, 1989.
- [7] J. Morris, M. Satyanarayanan, M. Conner, J. Howard, D. Rosenthal, and F. Smith. Andrew: A Distributed Personal Computing Environment. *CACM*, 29:184–201, March 1986.
- [8] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.
- [9] J. Ousterhout, Da Costa, H. Harrison, J Kunze, M. Kupfer, and J. Thompson. A Trace-Driven Analysis of the Unix 4.2 BSD File System. In *Proceedings of the 10th Symposium on Operating Systems Principles*, pages 15–24, December 1985.
- [10] John K. Ousterhout. Why Aren't Operating Systems Getting Faster As Fast as Hardware? In *Proceedings of the Summer 1990 USENIX Conference*, pages 247–256, June 1990.
- [11] H. Patterson, G. Gibson, and M. Satyanarayanan. A Status Report on Research in Transparent Informed Prefetching. *SIGOPS, Operating Systems Review*, 27(2):21–34, April 1993.
- [12] D. Presotto, R. Pike, K. Thompson, and H. Trickey. Plan 9, A Distributed System. In *Proceedings of the Spring 1991 EurOpen Conf.*, pages 43–50, May 1991.
- [13] R. Sandberg, D. Goldberg, S. Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network File System. In *Proceedings of the Summer USENIX Conference*, pages 119–130. USENIX Association, June 1985.
- [14] M. Satyanarayanan. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Trans. on Computers*, 39:447–459, April 1990.
- [15] Peter Skopp and Gail Kaiser. Disconnected Operation in a Multi-User Software Development Environment. In *Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 146–151, October 1993.
- [16] A. Smith. Cache memories. *Computing Surveys*, 14(3), September 1982.
- [17] R. van Renesse, A. S. Tanenbaum, and A. Wilschut. The Design of a High Performance File Server. *Proceedings of the IEEE 9th International Conference on Distributed Computing Systems*, 1989.

10 Author Information

James Griffioen is an Assistant Professor in the Computer Science Department at the University of Kentucky. He received a B.A. in computer science from Calvin College in 1985, and his M.S. and Ph.D in computer science from Purdue University in 1988 and 1991 respectively. He was the recipient of the '89-'90 USENIX scholarship. His research interests include high-performance distributed file systems, scalable distributed shared memory systems, and high-speed network protocols. His email address is griff@dcs.uky.edu.

Randy Appleton is a Ph.D student in the Computer Science Department at the University of Kentucky. He received his B.S. degree from the University of Illinois in 1989 and his M.S. from the University of Kentucky in 1992. His research interests are distributed file systems, operating systems, and databases. His email address is randy@dcs.uky.edu.

Operating-System Support for Distributed Multimedia

Sape J. Mullender*

Ian M. Leslie†

Derek McAuley†

Abstract

Multimedia applications place new demands upon processors, networks and operating systems. While some network designers, through ATM for example, have considered revolutionary approaches to supporting multimedia, the same cannot be said for operating systems designers. Most work is evolutionary in nature, attempting to identify additional features that can be added to existing systems to support multimedia. Here we describe the Pegasus project's attempt to build an integrated hardware and operating system environment from the ground up specifically targeted towards multimedia.

1 Introduction

Since the invention of electronic computers in the forties, every decade has been characterized by new ways in which they were used. In the fifties, people used sign-up sheets to reserve the computer for an hour's work; in the sixties batch processing was introduced; time sharing became pervasive in the seventies; the PC and networking came in the eighties; and now, in the nineties, we see the introduction of multimedia.

These days, every self-respecting computer vendor sells computers with some form of multimedia support. Some workstations now have cameras built into them, PCs come with multimedia applications, even game computers now make use of CD-I. From a research viewpoint, multimedia seems to be a solved problem; can't we see the wonderful demonstrations from every vendor?

We argue that the multimedia applications on most systems today are inflexible, they more or less take

over the machine and cannot be combined with other applications.

Multimedia, we claim, is only real if the different media are treated with equal respect. Audio and video should not be second-class media on which the only operations are capture, storage and rendering, but media that can be processed — analysed, filtered, modified — just like text and data. This processing should not be a privilege of dedicated operating-system processes, but should be possible to do, interactively, with ordinary applications.

Existing multimedia systems do not have this ability. For example, on typical PC platforms, multimedia applications run in real time but take over the machine; on Unix platforms, multimedia applications co-exist with other applications, but they hardly run in real time. Sometimes, dedicated hardware can capture and render multimedia in real time, but the data is far removed from the processor so that no processing is possible.

The value of audio and video depends critically on the ability both to process and to render them in real time. This is hard. The value of *interactive* audio and video additionally depends on being able to capture, process and render it with fraction-of-a-second end-to-end latency. This is even harder.

In the Pegasus project, groups at the University of Cambridge Computer Laboratory and the University of Twente Faculty of Computer Science are rising to the challenge of providing architectural and operating-system support for distributed multimedia applications.

Pegasus is a European Communities' ESPRIT¹ project which is now halfway through its three-year funding period.

The goal of Pegasus is to create the architecture for a general-purpose distributed multimedia system and

*University of Twente, Faculty of Computer Science, P.O. Box 217, 7500 AE Enschede, The Netherlands

†Cambridge University Computer Laboratory, Corn Exchange Street, Cambridge CB2 3QG, United Kingdom

¹The Pegasus Project is supported by ESPRIT BRA project 6586 and partially supported by the Cambridge Olivetti Research Laboratory and a grant from Digital Equipment Corporation.

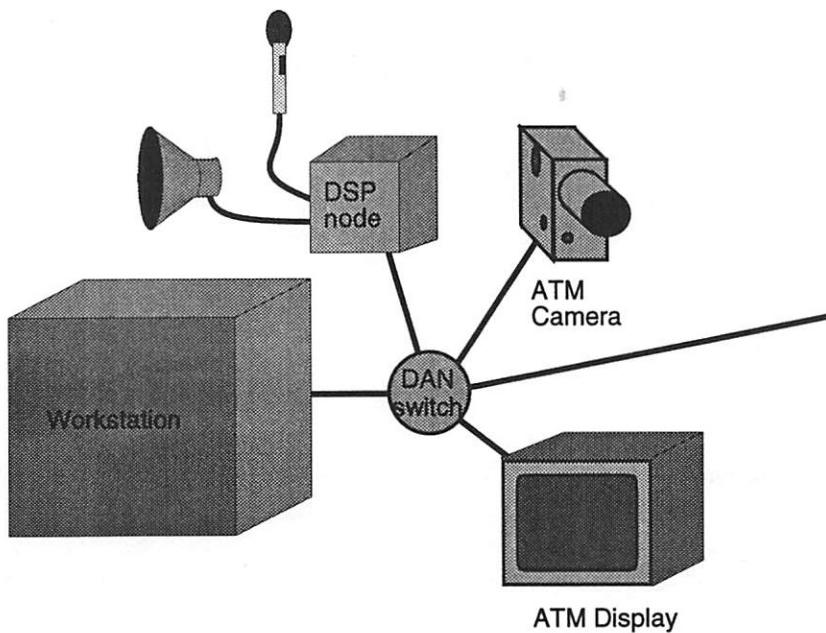


Figure 1: Architecture of the multimedia workstation

build an operating system for it that supports multimedia applications. A few specific applications will be implemented in order to prove the practicality of the system.

The architecture consists of: multimedia workstations; general-purpose and special-purpose multimedia processing servers; a single storage service for all types of data; and Unix boxes as the platform for the non-real-time control part of multimedia applications and applications unrelated to multimedia. All of the components are connected through an ATM network, which provides the bandwidth and can provide latency guarantees for interactive multimedia data. Multimedia capture and rendering devices are connected directly to this network, rather than being connected to, for example, workstation buses. This architecture is explained in Section 2.

The operating system support in Pegasus consists of a microkernel, named *Nemesis*, that supports a single address space with multiple protection domains, and multiple threads in each domain. There is scheduler support for processing multimedia data in real time. Nemesis has a minimal operating-system interface; it does not — at least, not now — have a Unix interface. However, processes on Nemesis can be created, be controlled by, and communicate with, processes on Unix. We expect multimedia applications to consist of symbiotic processes on Nemesis and Unix, where user interface and application control will be provided

by the Unix part, and real-time multimedia processing by the Nemesis part. Later, perhaps as part of another project, parts of the Nemesis functionality could be ported to a general-purpose operating system, or a Unix emulation provided over Nemesis. Nemesis is described in Section 3.

System services are viewed as objects: abstract data types accessed through their methods. When invoker and object share a protection domain, method invocation is through procedure call; when they share a machine, and thus an address space, invocation takes place through a protected call, or ‘local remote procedure call’; when they are on different machines, invocation goes via remote procedure call. Objects are located using a distributed name service. The name space is global only in the sense that every entity, in principle, can name any object in the universe; it is not global in the sense that there is one root to the name space, or that one name identifies the same object anywhere. Each protection domain contains a local name server which maintains connections with name servers elsewhere. The name server assists in establishing the appropriate channels through which local and remote objects are invoked. The name server is described in Section 4.

The Pegasus File Server is a log-structured file service designed to store and retrieve multimedia files in real time and to scale to a very large size. Scaling the file-server design up to terabyte capacity has forced

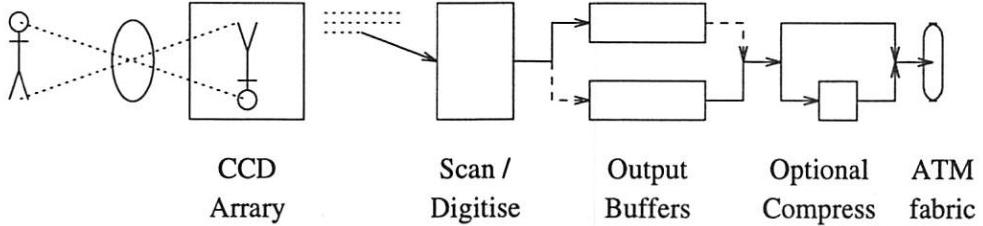


Figure 2: Principle of the ATM Camera

us to redesign the log-structured file-system structures as they occur in Sprite or BSD4.4. The Pegasus File Server uses a buffering and storage strategy that prevents loss of data in case of failure of a single component. The Pegasus File Service is described in Section 5.

2 Systems Architecture

In this section, we will show and explain the unusual architecture of the Pegasus system. The system consists of workstations and servers, interconnected by an ATM network. We use an ATM network as it can provide high bandwidth and low latency. ATM networks can scale gracefully to large sizes and link bandwidths and very large aggregate bandwidths.

Multimedia systems need special hardware for input and output of digital audio and video. Once digitized, video and audio streams must be transported to where they are processed, stored or rendered. Video requires substantial, but not staggering bandwidths: using frame-by-frame compression, for instance with JPEG, a video stream requires no more than a megabyte per second. Modern networks can easily provide this bandwidth. Using compression methods that compress groups of frames, such as MPEG, much higher compression can be reached, albeit at the cost of higher end-to-end latency. Audio has modest bandwidth requirements compared to video, but is much more susceptible to jitter, that is, the irregularities in the transport and processing times.

For smooth and efficient handling of interactive digital audio and video, the paths between origin and destination must be as short as possible. Gratuitous processing and transportation increase the end-to-end latency and hence decrease the quality. Thus, it is desirable that audio and video data are not handled by operating-system and application code except when application-specific processing is being carried out.

Figure 1 shows an important aspect of the Pegasus architecture — the target end-system architecture. The

figure shows a conventional workstation and its network interface connected to an ATM switch. However, also connected to the switch we see a camera device, a display device, an audio device, and then the rest of the ATM network. The important point is that the switch is under control of the workstation; that is, all connections through the switch are managed by the workstation, so that the workstation is also in control of the multimedia devices.

This setup is much like that of the *Desk-Area Network* (Hayter and McAuley [1991]). However, in a real DAN, an ATM switch fabric actually forms the central backbone of the workstation itself; CPU, memory and devices all communicate via the switch. The Pegasus project, partly because of its time frame of only three years, uses a conventional bus-based architecture for its processor devices, but uses the DAN mechanism for connecting multimedia devices.

In this architecture, when video flows from a camera in one system to a display in another — as is the case in video-phone and video-conferencing applications — no processors need to process any video data. This goes for the audio data too, of course. Hence the processors in the workstations, at both the camera and display, only need to manage the connections and devices.

2.1 Some ATM Devices

This section briefly describes the ATM devices used by the Pegasus project to provide a multimedia platform. More details of the DAN devices are available in Barham et al. [1994].

The ATM camera (Pratt [1993]), directly produces digital video as a stream of ATM cells. The principle of the ATM camera is schematically depicted in Figure 2. Scan-lines of video are digitized and when eight lines have been buffered, they are encoded as *tiles*, rectangles of 8×8 pixels. A number of tiles are packed into the payload of an AAL5 frame together with a trailer that provides the *x* and *y* coordinates of the tiles with respect to the video frame, and a time stamp that identifies the frame that the tile belongs to.

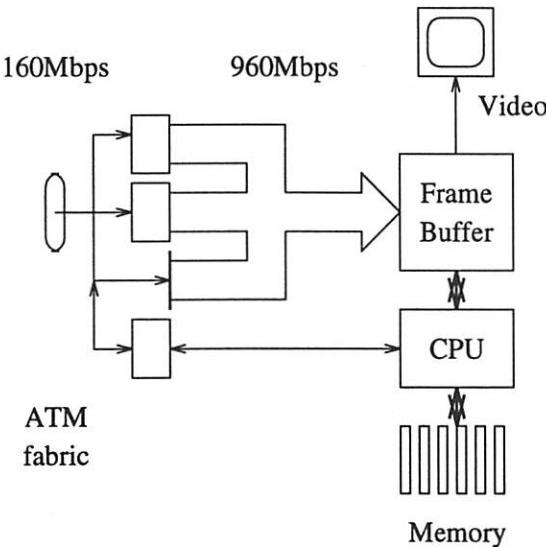


Figure 3: Architecture of the ATM display

Cameras can be equipped with one or more compression devices. The device to be used is identified when the virtual circuit is established. Currently, both raw video and motion JPEG are supported. Using AAL5 allows interaction with standard AAL5 implementations and offers protection against rendering or decompressing faulty tiles.

The version of the ATM camera now in production also includes audio capture capability.

The ATM display, shown in Figure 3, implements a single primitive, that of displaying arriving pixel tiles on incoming virtual circuits to windows on the screen. The virtual-circuit identifier (VCI) is used as an index into a table of window descriptors; each window descriptor has an *x* and *y* offset from the top-left-hand corner of the display, and clipping information. By manipulation of these contexts, a window manager can control which virtual channel, and thus which process, can access the different pixels of the screen.

Incoming data can be coded as compressed or uncompressed tiles. Note that as tiles essentially represent bit-blit operations of fixed size, from the viewpoint of a display, there is a unification of video and graphics. The code in conventional window systems that does the multiplexing of windows to the display can largely disappear; the multiplexing is done via the display's window descriptors. The window manager, exerting its control over the creation and modification of these descriptors, can create windows on screen, move them, resize them, iconize them and raise or lower them. It can also use a window descriptor that allows it to write the whole screen for decorating windows with title bars and resize buttons.

While the hardware for the display is under development, software emulation using a DS5000-25 is being used.

Finally, there is an ATM DSP node which combines digital signal processing and audio input and output. This device contains DACs and ADCs and packs and unpacks audio samples into ATM cells. Each such cell also contains a time stamp.

Our experience so far indicates that ATM devices are simple to construct and that they allow a natural combination of video data and graphic data on a display. The use of *tiles* for video reduces latency in several places from a 'frame time' (33 or 40 ms) to a 'tile time' (30 to 40 μ s). Since latencies tend to add up, this is an important reduction.

2.2 Control Protocol

Multimedia devices generate two streams of data on two distinct virtual circuits. One is the actual data stream which was cursorily described above. The other is a control stream; this is a bi-directional low-bandwidth stream that is used to control the device and for purposes of synchronization.

Both data and control virtual circuits are established through the normal mechanism of ATM signalling, although in the case of many of the ATM devices, this signalling is handled by a management process on the attached workstation, rather than by the device itself.

Typically, the device manager will connect the data stream directly to the sink or source; however, the control stream would normally be connected to a local synchronization process. For example, a host that wishes

to send synchronized audio and video, will do so by having the audio node and camera send the audio and video data streams separately (they have to end up in different devices too, at the other end), while a local process will merge the two control streams into a combined control stream for the playback control process at the rendering end. The playback control process is then responsible for the synchronization of the play-out of the various streams arriving at it, based on the source synchronization information from the remote manager(s) and data arrival events.

The Pegasus File Server, which can also be viewed as a multimedia device in this context, uses the control stream associated with an incoming data stream to generate index information that can later be used to go to specific time offsets into a media file or a set of synchronized files.

2.3 Systems Components

An overview of the Pegasus architecture is shown in Figure 4. In this figure, we can distinguish a Pegasus multimedia workstation, multimedia compute server, storage server and Unix server, all interconnected by an ATM network.

Each site is using locally developed ATM switches to provide the ATM network: the Fairisle switch in Cambridge (Leslie and McAuley [1991]), and the Rattlesnake switch in Twente (Smit [1994]).

The architecture of the multimedia workstation is as described above; multimedia input and output devices are connected to a local ATM switch (for which we use the Fairisle switch) and the rest of the workstation is entirely conventional. The multimedia processing nodes do not have special devices attached to them.

The multimedia workstations and processor nodes are controlled by a microkernel, called *Nemesis*. This kernel, which is discussed in more detail in Section 3, provides support for multimedia applications: timely scheduling and efficient interprocess communication.

One or more nodes in Pegasus run Unix. Applications on this platform have access to a rich collection of tools — compilers, text processors, graphics support, etc. — which, due to available effort, we do not intend to make available on the Nemesis kernel. We expect many multimedia applications to be split over Unix and Nemesis; the Unix part will contain the control functionality, whereas the Nemesis part will contain the necessary real-time functionality for audio and video processing.

This separation is entirely inspired by practical considerations. The Pegasus design team does not have the resources to add the kind of scheduling necessary for multimedia processing to existing operating-system

platforms, they are too big to modify². Separating Nemesis and Unix gives us the best of both worlds: a testable and measurable platform for multimedia applications and all the functionality of Unix. It is for another project to port Nemesis functionality to Unix or vice versa.

3 Kernel Support

The Nemesis kernel implements several unusual features, some of which are present to aid in the implementation of multimedia applications, others for the simple reasons of efficiency and tidiness. Here we summarize the major features.

3.1 Memory Model

A Nemesis kernel provides a number of distinct, schedulable entities, called *domains*. While all domains share the same virtual address space, privacy and protection are implemented using the appropriate access rights in the virtual address translations. Code executing within a domain may access memory within another domain only if both domains have explicitly arranged to share the memory.

Some examples highlight the approach: shared library segments would be mapped readable in every domain; a unidirectional inter-domain communications channel would be mapped read/write in the source and read-only at the sink; objects may be shared in shared read/write segments; etc.

The cost of using a single address space is the penalty of load-time relocation. We try to amortise this cost by caching the results of such relocations and then aim to reload an application at the same virtual address at which it was last executed. In this we are helped by the use of 64-bit VM architectures, which allow a sparse allocation of addresses so that we can arrange reuse with high probability. Consider for example allocating the top 32 address bits of a 64 bit virtual address based on a 32-bit hash function of the code to be executed.

The benefits of a single address space we are aiming for are: simplified sharing of data structures (in particular objects) between domains, and the removal of virtual address aliases which can result in significant context switch costs with caches accessed by virtual address.

3.2 Virtual-Processor Model

A domain differs from the normal concept of a user process in the way in which the processor is presented

²Yes, even Mach 3.0.

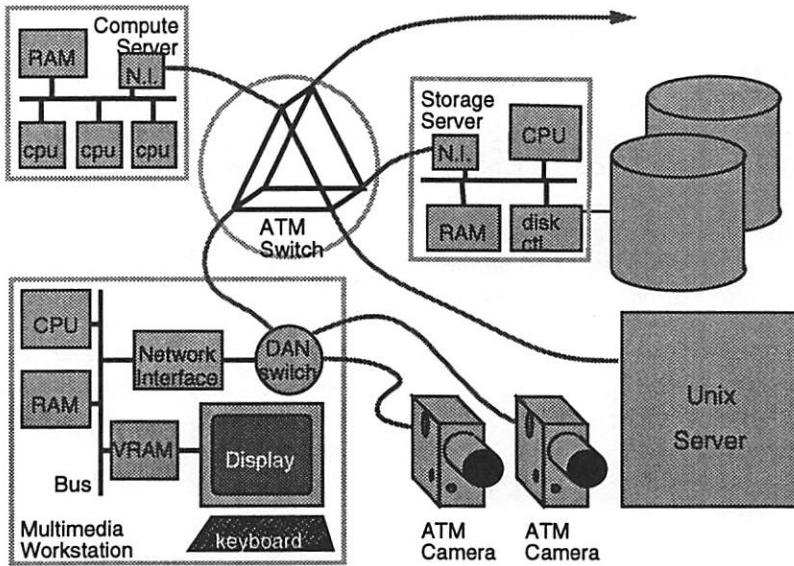


Figure 4: Pegasus architectural overview

to it. In the case of a process, the processor is taken away from it by *suspending* it and is returned by *resuming* the process to exactly the state in which it was when it was suspended. This gives the illusion to the process that it is running on its own *virtual processor*; it also hides from the process any information about the current processor availability — the process has no way of knowing *when* it has the processor.

In Nemesis, the processor is taken away from a domain by *deactivating* it; deactivation involves storing the state of the processor into a data structure shared by the kernel and domain, the Domain Information Block. When the domain is next scheduled, the processor is given to a domain by *activating* the domain; activation involves transferring execution to an address specified in the activation vector entry in the Domain Information Block.

For a domain supporting a traditional single-threaded model of execution, activation start up code would just restore the saved context and the user code would continue to execute. Another common use within the Pegasus project would be for the entry point to be a user-level thread scheduler. In this case the mechanism provides functionality similar to *scheduler activations* (Anderson et al. [1991]). Finally, some domains may be completely event driven, for example, device driver domains.

Hence it is simple to support the standard programming models on this activation model; in fact all operating systems do it, but it is usually the case that the asynchronous nature of interrupts and rescheduling events is hidden from the user level code.

The Nemesis mechanism provides a number of advantages for the types of multimedia applications we are considering. First, it provides a means of informing applications when they have the processor; a user-level scheduler can use this information, together with the current time, to make more informed decisions about the fate of the threads which it controls. Second, because thread scheduling is performed by the application, the user-level scheduler has direct control over the behaviour of its threads, and does not have to resort to describing their behaviour to a central scheduler in terms of priorities and deadlines. Third, once a domain is given the processor, it keeps it until its time quantum expires, or it voluntarily yields the processor because it has no more work to do. This avoids the problems encountered in kernel level thread implementations when threads block in the kernel and the kernel scheduler gives the processor which was running the blocked thread to a thread belonging to another process. Nemesis has no blocking system calls except “suspend” which will typically only be called by a domain user-level thread scheduler.

3.3 Domain Scheduling

To explain the scheduling mechanism adopted in Nemesis requires an understanding of how we see a flexible multimedia platform being used. The allocation of resources to applications will not be controlled solely by the applications themselves. Rather, we see users being able to control processor allocation much in the same way that they control pixel allocation in

window systems. Thus, applications will not always get what they want; they will have to adapt to the resources they are given. However, for a particular time, seconds or tens of seconds, some of the resources given to an application may be viewed as “guaranteed”. The application may choose to use an particular algorithm on the basis of this guarantee. It may also be able to exploit unguaranteed resources which become available fortuitously.

The approach to scheduling in Nemesis is to schedule domains with a weighted scheduling discipline, where the weights are calculated from the user’s current policy. Within a given time frame, not all domains may use their allocation; the policy for sharing out remaining resources is still the subject of investigation. While domains have some processor allocation remaining, the current scheduler implementation uses an earliest deadline first algorithm to select between them.

Above this primitive-level scheduler, and running on a longer time scale is a *Quality-of-Service*-manager domain whose task is to update the scheduler weights; this is performed not only in response to applications entering or leaving the system, but also adaptively as applications modify their behaviour — this is performed on a longer time scale than the individual scheduling decisions in order to smooth out short-term variations in load.

3.4 Events

Nemesis provides a single mechanism by which domains can communicate the occurrence of *events* to each other — this also includes indications from interrupt handlers. A domain is eligible for scheduling when it has pending events, at which point it is included in the scheduling mechanism described above. Then, when a domain is activated, it is informed of pending events.

Events themselves do not carry values, but merely indicate that something has occurred. This may be the updating of a shared object, the arrival of a message from the network, passage of time, etc.; however, closures (ie. methods and data) are associated with each event and hide this heterogeneity from the event dispatcher.

The examples of a protocol domain processing arriving packets and inter-domain procedure calls highlight the need for two types of event signalling: synchronous and asynchronous, depending on whether signalling an event should cause a domain to voluntarily give up the processor to the signalled domain or continue executing. In the inter-domain call example, implemented using a pair of message queues in shared memory be-

tween the relevant client and server domains and a pair of events, lowest latency for a client/server interaction will be achieved by the client and server implementing the synchronous form of notification. However, a domain performing demultiplexing of incoming packets may be most efficient using the asynchronous means.

3.5 Kernel Privileged Sections

Device drivers and other trusted modules need to be able to protect themselves against interrupts, have access to privileged instructions, etc., for some part of their operation. The code that requires this access is often a tiny proportion of the total module; however, most operating systems would require that the whole module run in kernel mode, whether linked statically or dynamically loaded. Furthermore, it becomes a property of the code that it runs in kernel mode, rather than the data the code is manipulating.

Nemesis offers the concept of the Kernel-Privileged Section to meet the requirement for a dynamic and extensible means to provide access to kernel mode. Privileged domains may define sections of their code which need to be executed in kernel mode. In a block-structured language this would naturally be a basic block enclosed with some form of TRY ... FINALLY construct allowing privileged code to raise exceptions but forcing the thread to leave kernel mode before any handler outside the privileged section is invoked (see Figure 5). The implementation of the Kernel-Privileged Section (i.e. the begin_KPS and end_KPS) is highly processor dependent — on 68k, MIPS and ARM processors it leads to various traps implemented in a non-procedural manner, while the aim on the Alpha is to implement a PAL instruction to achieve the desired effect.

In many ways the Kernel-Privileged Section idea is akin to using locked critical sections for currency control, whereas most other operating systems have a model of kernel mode access more akin to *monitored procedures*.

3.6 Nemesis State

A primitive form of the Nemesis kernel, *Nematode*, has been implemented on DECstation 5000 (Hyden [1994]); this provides domains, events, and scheduling support. Currently Nematode is being evolved to conform to the machine independent interfaces defined for the Nemesis kernel.

The VM model and communications abstraction are adopted from those used for Wanda (Dixon [1991]); migration of this code awaits completion of the Nemesis kernel.

```

... <unprivileged code>

begin_KPS();           /* enter privileged section */
TRY
    ... <privileged code>

FINALLY
end_KPS();             /* leave privileged section */
END;

... <unprivileged code>

```

Figure 5: Coding a Kernel-Privileged Section

4 Naming and Invocation

Most objects (entities, things) will be used locally. Therefore, most names of objects used will be names of local objects. Name resolution should, therefore, be most efficient for local names. This implies that local names should be shortest and suggests that names of local objects should normally be near to the root of the naming tree.

This, it must be clear, is a deviation from a trend towards using *global name spaces*. In a singly rooted global name space, the shortest path names refer to countries or organizations; it is rare that we wish to name those by themselves. The most widely claimed advantage of a global name space is that objects have the same name anywhere and that this facilitates sharing. What actually facilitates sharing much more is the proper use of naming conventions: One can often *guess* somebody's electronic-mail address, one looks for *T_EX* macro files in subdirectories of /usr/local/lib or /usr/lib, one gives C source code files a '.c' extension. If the conventions are disobeyed, programs fail.

By using naming conventions properly, one can create name spaces that are only global in the sense that any object anywhere can be named, but not necessarily by the same name everywhere. The root of the naming tree can be the most local object and longer path names generally name objects further away. Conventions must be used to allow object sharing and there is no reason why one convention could not be the use of a subtree named /global for global names.

This sort of naming is used in Plan 9 from Bell Labs. Pike et al. [1993] have already put forward some of the arguments for naming conventions being more important than global name spaces. Our naming mechanisms have been heavily inspired by those of Plan 9 as shall become clear.

Every process starts up with a built-in name space.

Usually, this name space is inherited from a parent process and is at least partly shared with other name spaces. The name space consists of a *local name space* which names objects local to the process, and *mounted name spaces* which name objects external to the process. The mount point of a mounted name space is a local object with a connection to a name space in another process. Name resolution in mounted name spaces takes place by making name-lookup requests through the connection to the other process. The result of this resolution is an object handle.

Using an object handle, objects can be accessed through their *methods*. The precise manner in which methods are invoked depends upon the "domain relation" between invoker and object. If they share a protection domain then the invocation is a procedure call; when they are in the same address space but different protection domains (for example on the same Nemesis machine) invocation is by protected call; and when in different address spaces invocation is performed by remote procedure call.

When making an invocation there is always code at the invoker's end that depends on the call interface. In the case of a local procedure call, this interface-dependent code is generated by the compiler. In the case of system calls it is loaded from a library and in the case of remote procedure call it is generated by a *stub compiler* and linked with the rest of the caller's code.

Client stubs for far-away objects may do more than just transport call parameters to the remote objects; they may, for instance, perform caching so that there is no longer a one-to-one mapping between client calls to the stubs and calls to the remote objects. Such intelligent stubs are referred to as *agents* or *clerks*.

When objects can migrate, for instance, to where they are accessed, the interfaces to them may change. This means that the interface with which calls are to

be made is not always known *a priori*; the calling code depends on where the object is found when it is invoked.

Early distributed systems solved this by using the most general invocation method always: remote procedure call. This is not an optimal solution, especially now that dynamic linking can be used to invoke optimal code for the kind of call to be made in the case at hand.

An object-naming mechanism can be used to make the mechanism whereby object-interface code is loaded transparent. In our model, the resolution of the name of an object results in a *handle*. This handle is essentially a pointer to the interface to the object. For our handles we use *maillons* (Maisonneuve, Shapiro and Collet [1992]), which consist of an opaque, fixed-size, object reference and a pointer to a function that returns the address of the interface when called with the reference as argument. The extra level of indirection provided by the maillon allows connections to objects to be set up, or objects to be fetched before their first invocation, but in the most common case — the object is already there and ready to be invoked — the maillon imposes very little overhead.

Object handles are first-class objects in that they can be passed as arguments in local and remote procedures. Passing an object handle for a local object to a remote process has the side effect of creating a connection through which the object can be invoked remotely.

The Pegasus remote-procedure-call mechanism is based on ANSA's RPC and layered on MSNA (McAuley [1989]). The Multi-Service Network Architecture is a protocol hierarchy for ATM networks that also caters for continuous-media transport.

5 Storage

The storage system in Pegasus is intended to store traditional file data as well as multimedia data efficiently. A storage service for multimedia data must have a large storage capacity (video produces half a megabyte per second compressed, so a half-hour video already occupies a gigabyte) and a guaranteed (fixed) service rate.

Ordinary data usually occupies less space and does not require a guaranteed service rate. The data rate does not have to be constant, but should be as high as possible. Locality of reference can be exploited by caching data in client and/or server memory. Most modern file systems demonstrate that caching yields substantial performance gains.

This applies to naming data too, albeit that directories can be cached more effectively when the semantics of directory operations are exploited in the caching al-

gorithms.

In contrast, caching video and audio is usually not a good idea. If the system can already guarantee the appropriate rate for a video or audio stream when it is not cached, caching it will only use up memory, but cannot result in a higher performance — a fixed performance is desired. To make matters worse, caching would often be counterproductive: Most video sequences and many audio sequences are larger than the cache, so, by the time a user has seen, or an application has processed, a video to the end, the beginning has already been evicted from the (LRU) cache.

Since different kinds of data require different treatment in our storage service, it was decided to make a hierarchical design for it, where a common bottom layer is responsible for reading and writing the data on secondary and tertiary storage devices and maintaining the storage structures on them. Above this layer, different service stacks can be built using specialized algorithms for particular kinds of data.

These service stacks can be partially or wholly mirrored in file-server agents on client machines. Thus, caching strategies, for instance, can be jointly implemented by corresponding layers of code in client and server machines.

The service stack for continuous data on the server has been designed to interact directly with the multimedia devices of Pegasus. As described in Section 2, continuous streams composed of several substreams (synchronized video and audio is a typical example) will cause several data streams and one control stream to be generated. The storage server stores the data streams and uses the control stream to generate indexing information. This information then allows reading synchronized streams from a particular point, and fast forward, reverse play, etc. of these streams.

The bottom layer of the Pegasus storage service is called the *core layer*. It manages storage structures on secondary and tertiary storage devices and carries out the actual I/O. Pegasus uses a log-structured storage layout as was exemplified by Sprite LFS (Rosenblum and Ousterhout [1991]).

The log is segmented in megabyte segments. Each segment is striped across four disks. A fifth disk is used as a parity disk and allows recovery from disk errors.

Normal file data ends up in the log similarly to Sprite LFS. Continuous data, however, is collected in separate segments, although their metadata (the *inodes* or *pnodes* as we call them) are appended to the normal log.

The speeds of modern disks are such that the overhead of seeks between reading and writing whole segments is less than ten per cent, so that a transfer rate of at least five megabytes per second per disk is possible on

high-performance disk hardware. Striping over four disks makes a total bandwidth of 20 MB per second possible. We have not been able to test this yet, since our ATM network runs only at a mere 100 megabits per second, just over 10 MB per second.

Partly as a consequence of storing multimedia data, we have to expect that our storage service will grow large. We have set ourselves the goal to make the storage-service algorithms scale to a system size of 10 terabytes. Cleaning³ algorithms for a storage service of this size have to be designed carefully. If any part of the cleaning process scales with, say, the square of the system size, cleaning a terabyte file system will take a very long time.

We are currently implementing a cleaning algorithm whose complexity only depends on the number of segments to be cleaned and the amount of ‘garbage’. Roughly, it works as follows. During normal operation of the file system, the core maintains a *garbage file*. Every time a client write or delete operation creates garbage, an entry describing the hole in the log that corresponds to the obsolete data is appended to the garbage file.

When the file system needs to be cleaned, the garbage file is read and its entries are sorted by segment number. Then, a single pass through the garbage file is needed to find and clean all segments containing garbage. When cleaning is complete, the garbage file is truncated to a single entry describing the old garbage file itself.

Allowing client operations to continue during cleaning does not complicate the cleaning algorithm. At the start of a cleaning operation, the current place in the garbage file must be marked and cleaning uses only information before the marker while new garbage is appended after it. When cleaning is complete, the portion of the garbage file before the marker is deleted.

The first prototype of the core of the Pegasus file server now runs, with an incomplete cleaning mechanism. Higher-level services are being added; a Unix *v-node* interface is installed which allows the storage system to be used as a Unix file system.

Since files are stored on RAID, recovery from disk failures is straightforward. Once files have reached the disk, it is unlikely that they will be lost in a crash. Files, therefore, should be put on disk as soon as possible after they are written by the application. However, from a performance viewpoint, disk writes should be delayed so that overwrite operations and delete operations can be exploited to save disk operations. In the Pegasus

³Cleaning in a log-structured filing system is the act of recovering space which holds out of date information. Information may become out of date either because a later copy has been written or it has been logically deleted.

storage service we have tried to get the best of both worlds.

For this, we make use of the assumption that client and server machines crash independently. When an application makes a write operation, the client agent sends the data to the server and keeps a copy of the data in its buffers. When the server receives the data, it acknowledges this to the client agent which, in turn, unblocks the application. The data is now safe under single-point failures: when the server crashes, the client agent notices and either writes the data to an alternative server or waits for the crashed server to come back up; when the client machine crashes, the server will complete the write operation.

When there is a power failure, client and server will crash together. To guard against this, the servers can either be equipped with battery-backed-up memory, or with an uninterruptible power supply. With the latter, when a power failure occurs, the server has time to write its volatile-memory buffers to disk and halt.

These mechanisms obviate the need for writing data to disk quickly. For normal file traffic, this is not only beneficial for write performance — Baker et al. [1991] showed that 70% of files are deleted or overwritten within 30 seconds — but also for cleaning performance: The data that does eventually get written to the log is reasonably stable, so garbage is created at a much lower rate.

6 Conclusions

The Pegasus project reflects our belief that if distributed multimedia is to be supported effectively, a holistic approach to system design is required. Multimedia is not just a bolt on; it requires a fundamental reexamination of most aspects of the infrastructure. We have thought carefully about integrating multimedia devices into the network architecture of the system, we have looked at the data paths from camera lens to display screens, and we have analysed storage infrastructures from a performance, reliability and consistency perspective.

Thus far we have found that this approach gives a clean system design and makes our implementations efficient and simple. The desk-area network as the connecting infrastructure for machines and devices has greatly simplified the architecture of the rest of the system.

In the storage service, we have discovered that techniques for consistent caching, data buffering, log structure and RAID, each of which, by itself, is difficult to integrate in an existing environment, can be combined in a new storage system architecture. Consistent caching, buffering and RAID gave us reliability (no

data loss in a single crash); log structure and RAID give us good write performance.

Pegasus is only half-way through its funding period now and a lot of work still needs to be done. We hope we can demonstrate a complete system in two years' time. The results of our project are naturally public and we intend to make all code available where it is not restricted by licences from others.

7 Acknowledgements

The work presented here is the work of a team whose members we gratefully acknowledge: Paul Barham, Ralph Beckett, Richard Black, Peter Bosch, Tatjana Burkow, Shaw Chaung, Simon Crosby, Feico Dillema, Richard Earnshaw, Dave Evers, Robin Fairbairns, Daniel Gordon, Mark Hayter, Paul Havinga, Eoin Hyden, Pierre Jansen, George Linnenbank, George Neville-Neil, Ian Pratt, Timothy Roscoe, Paul Sijben, Gerard Smit, Martijn van der Valk, Lars Vognild, and Lex Warners.

8 References

- Anderson, T. E., Bershad, B. N., Lazowska, E. D. and Levy, H. M. (October 1991), Scheduler Activations: Effective Kernel Support for the User-Level Thread Management, *Proceedings of the 13th Symposium on Operating Systems Principles*, Pacific Grove, CA, In *ACM Operating Systems Review* 25(5).
- Baker, M. G., Hartman, J. H., Kupfer, M. D., Shirriff, K. W. and Ousterhout, J. K. (October 1991), Measurements of a Distributed File System, *Proceedings of the 13th Symposium on Operating Systems Principles*, Pacific Grove, CA, In *ACM Operating Systems Review* 25(5).
- Barham, P. R., Hayter, M. D., McAuley, D. R. and Pratt, I. A. (1994), Devices on the Desk Area Network, *submitted to Journal on Selected Areas in Communications*.
- Dixon, M. J. (September 1991), System Support for Multi-Service Traffic, University of Cambridge, Ph.D. Thesis.
- Hayter, M. and McAuley, D. (October 1991), The Desk-Area Network, *ACM Operating System Review* 25(4), 14–21.
- Hyden, E. A. (February 1994), Operating System Support for Quality of Service, University of Cambridge, Ph.D. Thesis.
- Leslie, I. M. and McAuley, D. R. (September 1991), Fairisle: An ATM Network for the Local Area, *ACM Computer Communication Review* 21(4).
- Maisonneuve, J., Shapiro, M. and Collet, P. (Oct. 1992), Implementing references as chains of links, *1992 Int. Workshop on Object Orientation and Operating Systems*, Dourdan (France), IEEE Comp. Society Press, 236–234.
- McAuley, D. R. (September 1989), Protocol Design for High-Speed Networks, University of Cambridge Computer Laboratory, Ph.D. Dissertation, Cambridge CB2 3QG, United Kingdom, Also available as University of Cambridge Computer Laboratory Technical Report No. 186, January 1990.
- Pike, R., Presotto, D., Thompson, K., Trickey, H. and Winterbottom, P. (April 1993), The Use of Name Spaces in Plan 9, *ACM Operating System Review* 27(2), 72–76, Reprint from Proceedings of the Fifth ACM SIGOPS European Workshop, Mont Saint-Michel.
- Pratt, I. (Feb. 1993), ATM camera V1, in *ATM Document Collection 2 (The Orange Book)*, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CN2 3QG England, 28.1–28.7.
- Rosenblum, M. and Ousterhout, J. (October 1991), The Design and Implementation of a Log-Structured File System, *Proceedings of the 13th Symposium on Operating Systems Principles*, Pacific Grove, CA, 1–15, In *ACM Operating Systems Review* 25(5).
- Smit, G. J. M. (February 1994), The Design of Central Switch Communication Systems for Multimedia Applications, University of Twente, Faculty of Computer Science, Ph.D. Thesis.

Splicing UNIX into a Genome Mapping Laboratory

*Lincoln Stein, Andre Marquis, Robert Dredge, Mary Pat Reeve,
Mark Daly, Steve Rozen, Nathan Goodman
Whitehead Institute for Biomedical Research
One Kendall Square
Cambridge, MA 02139*

Abstract

The Whitehead Institute/MIT Center for Genome Research is responsible for a number of large genome mapping efforts, the scale of which create problems of data and workflow management that dictate reliance on computer support. Two years ago, when we started to design the informatics support for the laboratory, we realized that the fluid and ever-changing nature of the experimental protocols precluded any effort to create a single monolithic piece of software. Instead we designed a system that relied on multiple distributed data analysis and processing tools knit together by a centralized database. The obvious choice of operating systems was UNIX. In order to make this choice palatable to the laboratory biologists—who rightly consider it their job to do experiments rather than to interact with computers, and who have come to expect all software to be as intuitive and responsive as the Apple Macintoshes on their desks—we designed a system that runs automatically and essentially invisibly. Whenever it is necessary for the informatics system to interact with a member of the laboratory we have carefully chosen a user interface paradigm that best balances the user's expectations against the system's capabilities. When possible we have chosen to adapt familiar software to our user interface needs rather than to write user interfaces from scratch. We've managed to hide the power of UNIX behind the innocuous personal computer-based front ends our users know and love, using techniques that should be applicable in other environments as well.

1. Introduction

The Whitehead Institute/MIT Center for Genome Research (WI/MIT CGR) carries out large-scale genome mapping projects. A genome map is composed of a large number of short DNA sequences called "markers" which have been ordered and

assigned to unique positions on chromosomes [National Research Council, 1988]. The availability of such maps greatly simplifies the task of identifying and isolating genes relevant to the understanding of development and disease. There are two main genome mapping projects at the WI/MIT CGR: the creation of a genetic map of the mouse [Dietrich *et al.* 1992], and the creation of a physical map of the human [Green and Olson, 1990]. We estimate that these projects will require the completion of several million individual experimental steps. This paper describes the design of the WI/MIT CGR informatics system and the lessons we learned during the process.

1.2 Choosing UNIX

Managing information flow in laboratory projects of this scale presents several challenges. The first challenge is managing the laboratory data for each project. The second is data analysis. The third is managing the dissemination of information both within and outside the laboratory. In addition there is a meta requirement: biomedical research protocols are a moving target. Laboratory techniques are constantly improving, and major and minor adjustments of the experimental protocols occur on a regular basis.

We chose to base our informatics system on the UNIX operating system for several reasons. First, a large number of UNIX utilities for analyzing molecular biology data already exist in the public and commercial domains. Second, UNIX is an open system that is available on many different platforms and is familiar to the academic world. Finally, the tool-based philosophy of UNIX [Kernighan and Plauger, 1978], with its emphasis on inter-process communication, lends itself to a modular design. We felt that the use of multiple modular data analysis and processing tools instead of a single monolithic piece

of software would allow us to respond more promptly to changes in the laboratory protocol.

However, the choice of UNIX, rather than a PC-based operating system such as MS-DOS, OS/2 or the Macintosh OS presented user interface problems. The laboratory scientists are familiar with personal computers, primarily the Apple Macintosh, and expect software to behave as it would on a desktop system. We could not reasonably expect biologists in the laboratory to master a series of data analysis tools running under an unfamiliar operating system.

1.3. The Laboratory Protocol

A flowchart of the mouse genetic mapping protocol is shown in Figure 1. The aim of the protocol is to

obtain small random DNA sequences that contain simple sequence repeats, such as $(CA)^n$, flanked on both sides by nonrepetitive sequences. These sequences are useful because they are frequently "polymorphic" between inbred mouse strains. In other words, the length n of the repeat varies between two or more strains of mice. Like eye or hair color, the length of the repeat is a genetic trait that is transmitted from one mouse to its progeny, and like other genetic traits, it can be mapped to a particular position by performing a series of genetic linkage analysis experiments.

The protocol begins by creating a library of mouse DNA sequences. This is done by cutting up whole mouse DNA into small pieces and inserting them into a self-replicating virus. Each viral clone in

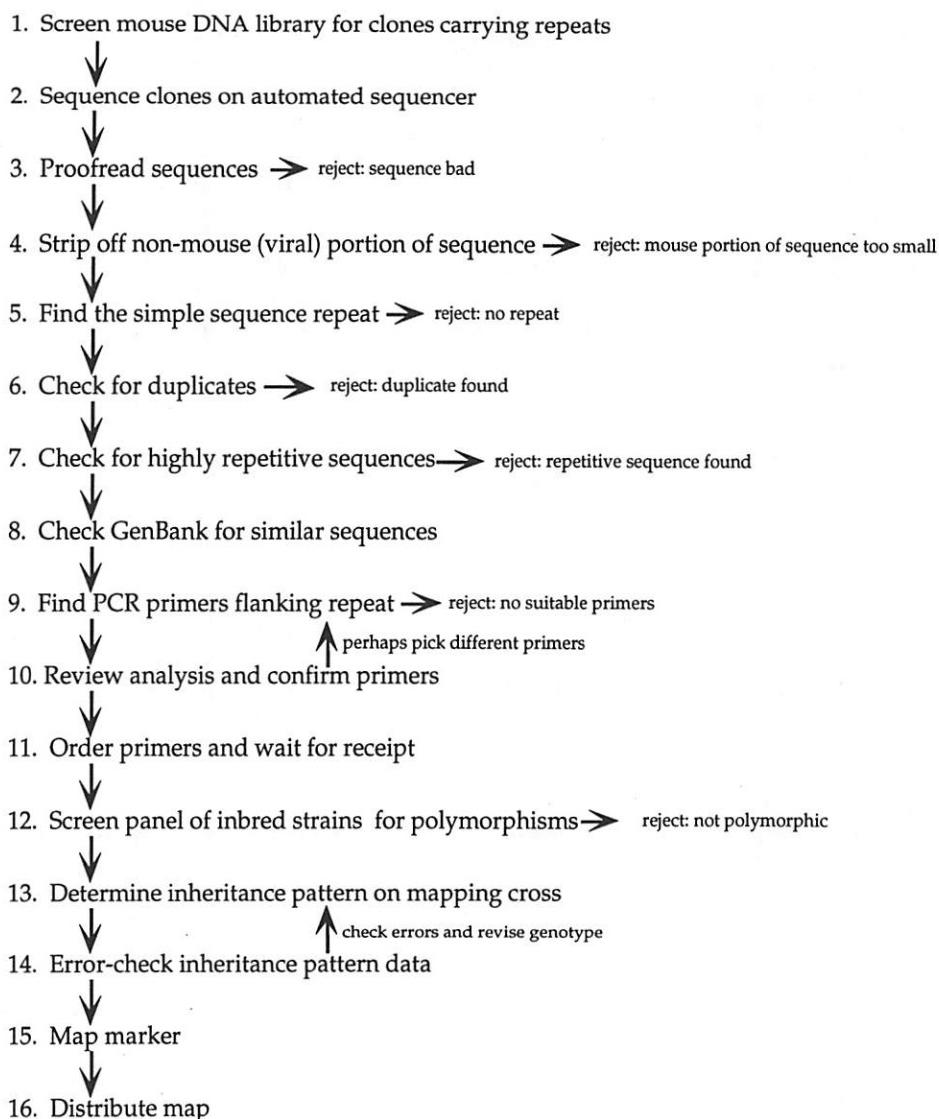


Figure 1: Genetic Mapping Protocol

the library contains a different random piece of mouse DNA. These clones are individually tested to find those that are likely to contain repeats, and each likely candidate is sequenced on an automated sequencing machine. Next comes a series of sequence analysis steps. First, since each clone consists of a mixture of viral and mouse DNA, the known viral portion of the sequence must be found and conceptually stripped off. Next, the sequence is scanned to identify the simple sequence repeat, if any. After this, the sequence is checked for duplicate sequences already present in our database and for the presence of highly repeated sequences that are present multiple times in the genome. If the sequence survives these tests, it is next compared to all entries in the GenBank database of published sequences in order to determine whether this sequence has ever been seen before. Finally, we choose a pair of short (about 20 base pair) sequences on either side of the simple sequence repeat to serve as "primers" for a biochemical technique known as the polymerase chain reaction (PCR). PCR allows us to rapidly determine the length of the simple sequence repeat without tediously recloning and sequencing it.

After having a biotechnology supply company synthesize the primer pairs, we characterize the simple sequence repeat further. Using PCR we determine the lengths of the simple sequence repeat in DNA taken from 12 common inbred mouse strains. Those repeats that are not polymorphic, that is, that do not vary in size between strains, are discarded. Those that are polymorphic are subjected to genetic mapping experiments that determine the length of

each of the repeats in the offspring produced by mating two of the inbred strains. Repeats that are close together on a chromosome will tend to remain together when inherited – they will appear to be "linked" – while those that are further apart or on different chromosomes entirely will be inherited independently of each other. The genetic mapping data is now fed into a program which does the number crunching necessary to order each of the markers and determine the distance between them.

The genetic mapping protocol is essentially a data pipeline. At any given time approximately 600 sequences are in the midst of processing. Experiments can fail and need to be redone, or sequences can be found to be unsuitable and be dropped from the pipeline at various steps. While managing the protocol certainly requires ingenuity in data modeling and data processing, we have found one of the most challenging tasks to be integrating the Apple Macintosh-based work habits of the laboratory with our tool-based philosophy of UNIX.

2. Informatics System Architecture

The architecture we have chosen is diagrammed in Figure 2. The major features are:

- A centralized database running on a UNIX workstation that stores the experimental results of all steps in the mapping protocol. Our database is called "MapBase" [Goodman *et al.* 1993, Goodman 1994] and is an object-oriented database written in C++. It is a multiconnection client/server database that

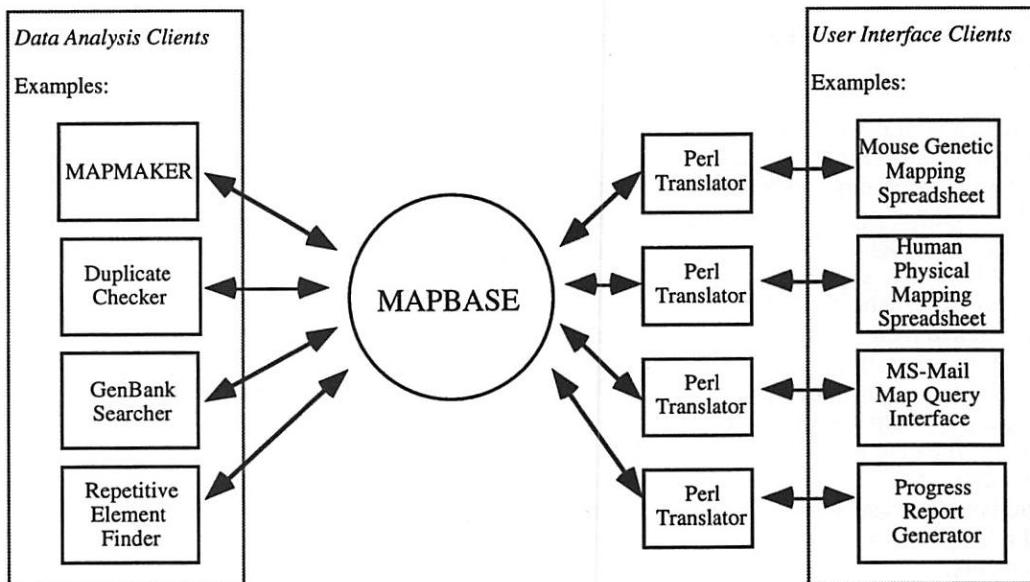


Figure 2: Informatics system architecture

supports a simple query and update language and is accessible over the TCP/IP network. An important design decision was to make MapBase accessible only to software tools and to programmers. Our end users—scientists and laboratory personnel—never interact with the database directly.

- Data analysis and manipulation clients that run automatically on UNIX workstations, often under the UNIX *cron* utility. The system is decoupled so that the database and data analysis clients do not have to reside on the same computer. Scientists do not need to take any action to initiate the data processing, but the system keeps them apprised of experimental progress by E-mailing status reports.

- Whenever human interaction with the informatics system is necessary, it is done through familiar spreadsheet, E-mail, and word-processing software running on the same personal computer the laboratory personnel use for other aspects of their work, the Apple Macintosh. This off-the-shelf software provides scientists with comfortable interface paradigms for interacting with the informatics system. By carefully matching the paradigms to the tasks, we have been able to match user expectations with the system's capabilities.

- The MapBase database uses a text-only query and transaction language which was designed to be easily machine-parseable. While many of our data manipulation clients interact directly with MapBase, most of them, particularly the adapted Macintosh programs, communicate via intermediary perl scripts [Wall and Schwartz, 1991] which handle the translation between MapBase and the client.

2.1 Choice of User Interface Paradigms - Lessons We Have Learned

Our first pass at creating user interfaces between the laboratory and the informatics system was disastrous. Adopting the conventional approach, we wrote a number of specialized applications for the graphical input and display of laboratory data. To our chagrin, biologists stubbornly kept their primary data in various Excel spreadsheets that they had created themselves, transferring the data to the informatics system in large batches only when absolutely necessary. They requested that we make our graphical data entry forms more spreadsheet-like, and were unhappy when we were unable to reproduce the full functionality of Excel. They tired of waiting for our interactive database queries to complete and developed a habit of E-mailing their data requests to the programmers.

Our second attempt to create user interfaces took a different approach. Rather than build user interfaces from scratch, we took the software and data management techniques the lab members were already using and modified them to work with the informatics system. In contrast to the first attempt, these interfaces won immediate acceptance and are still in use today. What follows are specific examples of laboratory user interfaces we have created and the general lessons we have drawn from this experience.

2.1.1 The best user interface is no interface at all

From the laboratory's point of view, computers are at their best when they work invisibly behind the scenes. When possible we have made our data processing steps invisible and automatic. For example, every new DNA sequence that is entered into MapBase needs to go through a series of software checks and characterizations: the sequence has to be checked against other sequences in the database to catch duplicates. If it passes this test, the sequence is checked against worldwide DNA sequence databases to see if it has already been mapped. Next the sequence is examined for the presence of repetitive elements (sub-sequences known to be present multiple times in the genome), and so on. Rather than ask a member of the laboratory or (heaven forbid) a programmer to initiate these tasks, the informatics system does it automatically. Every night a *cron* job queries MapBase for new DNA sequences, and feeds any that are found through a series of small programs that perform each of the characterization steps. These programs are, in fact, implemented as a series of filters connected together through UNIX pipes. The program at the very end of the pipeline gathers up the results, feeds them back into the database, and E-mails a status report to the scientists in the laboratory.

The sequence characterization programs are a nice illustration of the UNIX toolkit approach. The input for each of the programs is a series of keyword/value attributes in the format KEYWORD=VALUE. Keywords identify the sequence and the information collected on it in previous steps. Programs in the pipeline extract the attributes in which they are interested, pass through the rest, and add any information that they wish to contribute. For example, the program that determines the start and length of the simple sequence repeat looks for the following attributes in the data stream

```
NAME=MJ100
SEQUENCE=GATTGACGAGATCACAGTTGGCACAC
ACACACACACCAAGTTGAATTTCCTGG
```

and adds to it the following (passing through the name, sequence and any other attributes present):

```
REPEAT_START=22  
REPEAT_LENGTH=20
```

The particular order in which sequence processing steps are performed is determined by a shell script, which makes it easy to rearrange the order in which processing is performed, insert data processing modules, or experiment with alternate algorithms.

Another example of invisible processing is the program that assembles genetic maps. Every night new mapping data is incorporated into the growing genome maps by a *cron*-invoked shell script and the newly constructed maps are then E-mailed to the scientists. In UNIX style, the data processing steps are performed by separate programs. A large C program called MAPMAKER [Lander *et al.* 1987], does the actual multipoint genetic linkage analysis. A second program interprets the numeric output from MAPMAKER and converts it into graphical maps, while a third converts the maps into Macintosh PICT documents and E-mails them to the scientists.

2.1.2 For moving large amounts of data use "drop folders"

There are times when it is necessary for data to be fed into the informatics system in large chunks. One example of this is the entry of raw DNA sequence information into the database. Automated DNA sequencing machines produce this data in the form of Macintosh text files. The challenge is getting these files from the Macintosh into MapBase. Rather than asking laboratory members to cut and paste these files into a specialized data entry program or to use a UNIX utility such as *ftp*, we have designated a "drop folder" on a disk that is cross-mounted between the UNIX and Macintosh systems. This disk appears as a NFS volume to the workstations and as an Appleshare volume on the Macintosh desktops. To transfer the sequence files to the UNIX system the user just drags them into the drop folder. A *cron*-launched perl script checks this folder periodically for new files, reformats them, and feeds them into MapBase.

2.1.3 To represent tabular data use a spreadsheet.

There are situations that require more give and take

	A	B	C	D	E	F	G	H	I
1	Box	Primer Pair	ob	cast	spr	A/J	B6	C3H	DB
2	140	MT1937	160	-1	160	160	-1	-1	-1
3	140	MT1938	138	164	134	138	138	138	150
4	140	MT1939	108	112	-1	108	108	108	110
5	140	MT1940	98	132	94	96	98	96	98
6	140	MT1941	-1	-1	-1	-1	-1	-1	-1
7	140	MT1943	146	138	178	146	146	146	146
8	140	MT1944	-1	-1	-1	-1	-1	-1	-1
9	140	MT1945	130	132	114	108	130	108	130
10	140	MT1946	114	138	138	114	114	104	114
11	140	MT1948	96	88	93	82	96	82	96
12	140	MT1949	148	168	160	146	148	148	148
13	140	MT1951	126	130	138	132	126	132	132
14	140	MT1953	168	170	145	146	168	146	146
15	140	MT1955	132	146	154	146	132	132	138

Figure 3: Portion of spreadsheet used in data entry for mouse genetic mapping protocol

between the computer and the scientist than is offered by either behind-the-scenes processing or one way data transfer. For these situations we make extensive use of Microsoft Excel for data entry, viewing and manipulation. The spreadsheet has become a familiar and intuitive user interface, and is in fact the way that most of the scientists in the laboratory are accustomed to storing and organizing laboratory results.

We have created a set of external code modules ("plug-ins") that give Excel network access to MapBase and other UNIX tools. With the appropriate macros, these plug-ins allow custom spreadsheets to write, retrieve and update MapBase records. In this way a scientist can open up a spreadsheet that lists a subset of the sequences and the experimental results associated with them. Although the data appears to be local to the spreadsheet and can be copied, pasted, summarized, printed and otherwise manipulated in the usual spreadsheet manner, it is actually tied to the underlying data structures in MapBase. To add or edit data, the scientist makes the appropriate changes in the spreadsheet and chooses the "Send to database" menu command, which updates MapBase. A screenshot of one of our spreadsheets in action is shown in Figure 3 (previous page). This spreadsheet is used to enter the lengths of simple sequence repeats in various inbred strains.

In addition to the advantages of familiarity and ease of use, this approach has allowed us to incorporate data analysis and data integrity checking tools to the Excel spreadsheets in a simple and extensible manner. The most frequently-used Excel plug-in simply sends the contents of the spreadsheet as tab-delimited text over a TCP/IP socket to a waiting perl script. The perl script figures out what's to be done with the spreadsheet data, invokes the appropriate data analysis and database tools, adds or modifies the text and sends it back over the socket to the Excel plug-in which obligingly pastes it back into the spreadsheet. Adding new behavior to the spreadsheet is often as simple as updating the perl script.

A good example of this extensibility is our experience when we decided to add error checking to the genetic linkage mapping data spreadsheet. A frequent source of laboratory error occurs when laboratory technicians make typographical errors when they type the simple sequence repeat length inheritance patterns into the spreadsheet. The error is caught that night when a full MAPMAKER run is performed and the repeat is found to be unmappable, but by this time the data has already been entered into MapBase and the technician has put the experimental

results away. We wished to catch errors at the point of data entry, while the experimental results were still fresh. While it was impractical to invoke a full MAPMAKER for each new simple sequence repeat entered, it was possible to write a quick and dirty program that roughly maps new repeats and catches most errors. By having the perl spreadsheet listener invoke this error-checking program, we were able to give technicians rough mapping feedback almost immediately and to catch the typographical errors before the data was entered into MapBase. Best of all, we did this without writing any new Macintosh code.

2.1.4 For database queries, use an E-mail interface.

E-mail is a particularly effective paradigm for posing database queries. Complex MapBase queries can take 20 to 30 seconds to complete. While this is not a particularly long wait, it is too long for an interactive session mediated by a graphical user interface, where an immediate response is expected. In contrast, scientists are accustomed to using E-mail to query each other, and they expect a delay of minutes to hours between sending out a request for information and receiving a response. Scientists can address *ad hoc* queries to MapBase via familiar E-mail software, Microsoft Mail, using a series of graphical forms we've designed. By filling out checkboxes and text fields, users of the system can set the conditions to satisfy and select the data fields to retrieve. They then send the form to the database and in less than a minute their query is answered by return mail. Textual data, such as status reports, is returned as word processor documents, while graphical information, such as maps, is returned in the form of Macintosh picture files. Tabular data, as one would expect, is returned as spreadsheet files. This E-mail system is also used for posing queries to MapBase over the Internet using a set of text-only forms. Figure 4 (next page) shows a portion of one of our E-mail forms. This one is used within the laboratory to obtain information about the progress of the genetic mapping protocol.

The E-mail query system is implemented using familiar UNIX tools. An alias called "genome_database" pipes incoming mail to a perl script that determines which query form is being used and where the query is coming from. Using this information, the perl script reformats the query form into a series of keyword/value pairs. This data is then passed to another perl script that queries MapBase and reformats the results in human-readable form. In some cases, when the user wishes to receive data as a spreadsheet or a picture file, the result text goes through an addition step in which it is piped through UNIX tools that reformat it as appropriate (these

tools are written in C or perl). Finally the data is E-mailed back to the user.

The translation of queries from the graphical format used by Microsoft Mail to the text-format expected by the UNIX E-mail query system is accomplished by Microsoft Mail itself. Graphical forms sent outside the Microsoft Mail system are converted into a textual representation using rules that can be specified when the forms are designed. We specify conversion rules that are easily perl-parseable.

2.1.5 When all else fails, do it yourself.

We did of course encounter a small number of situations in which existing Macintosh software did not handle the job. The two examples that we encountered both involved entry of laboratory image data. In one case the solution was to capture the image via a video camera and interpret it using custom image analysis software. In another case the

solution was to use a digitizing tablet to capture the positions of data points using a small Macintosh program and then to automatically paste the data into an Excel spreadsheet. Both these tasks were made easier by the use of Apple Events, which allow Macintosh programs to exchange data and coordinate their activities. In the case of the former task, the software that controls the video camera resides on a Macintosh while the software that interprets the image data resides on a DEC alpha/OSF 1. To integrate the two, we wrote a Macintosh daemon process that listens on a TCP/IP port for incoming messages from the image analysis program and forwards them, in the form of Apple Events, to the camera control program.

3. Conclusions

We have found the UNIX toolkit approach to be very helpful in designing and maintaining our informatics system. As the laboratory protocol has changed,

The screenshot shows an E-mail message window with the subject "Discarded Sequences". The message body contains several input fields and checkboxes:

- To:** Database Query
- Subject:** Discarded Sequences
- Data Requested:**
 - Sequence
 - BLAST matches
 - Status
 - Primers
 - FASTN matches
 - Summary
 - Allele Sizes
 - Duplicates
 - GENOTYPE ERROR FORM
 - Genotype
 - Chronology
 - BOX SUMMARY SHEET
- For Markers Named:** (name1..name2 = range, ~name = partial name)
- For Data Entered From (Date):**
- To:** [empty field]
- Box Number(s):** [empty field]
- Sequenced by technician:** [empty field]
- For Markers Polymorphic in Cross** [empty field] X [empty field]
- At a threshold of** 2 bp

On the right side of the message body, there is a sidebar titled "STS PIPELIN" containing the following settings:

- Organism:** Mouse (radio button selected)
- Limit Search To:**
 - any marker
 - linked marker
 - unlinked marker
 - ready for PR
 - ready for pri
 - ready for scr
 - ready for ger
 - interesting B
 - junked sequen
 - those with pr
 - those with pr
- Sort Results By:**
 - no particular
 - chronological
 - assay name
 - sequence narr

Figure 4: Portion of an E-mail form used for querying MapBase

we've been able to modify our system by adding or altering software modules or changing the order in which they execute. We have adopted the same approach to the design of our user interfaces. Instead of building our own interfaces from scratch, we have adapted existing software with which our users are already comfortable. The result has been a system that has allowed data throughput to grow by a factor of six (from 50 genetically mapped sequences per month to over 300/month) over a period of a year, and that enjoys a high level of user satisfaction.

Our approach is different from those taken by several other genome mapping groups. One approach, exemplified by the ACEDB database of the *C. elegans* genome is the "laboratory notebook" approach [Sulston *et al.*, 1992], in which the entire user interface is incorporated into one custom piece of software. An advantage of this system is that the interface is carefully crafted to fit the application. A disadvantage is that it is difficult to adapt the interface to meet changing laboratory needs. A more tool-oriented approach has been taken by the Chromosome 11 project, in which a relational database is used to integrate and control the activities of a number of data analysis and manipulation tools [Clark *et al.* 1994]. However in this case the decision was made to use a Apple Macintosh-based relational DBMS in order to take advantage of that system's graphical user interface, and this design decision has restricted the possibilities for automating information flow since the Macintosh OS does not provide the level of inter-process communication offered by UNIX. An approach similar to ours, but for a system to support a large-scale expressed sequence project, has been described by Kerlavage [Kerlavage *et al.* 1993]. They also build a pipeline of UNIX-based data manipulation tools drawing on a centralized database and interacting with scientists through Macintosh front ends. The major difference between their approach and ours is that their Macintosh interfaces are built on a single environment, Hypercard (Apple Computer), whereas we use different applications to present differing user interface paradigms.

Although our system was designed to meet the needs of a particular genome laboratory, our approach may be applicable in other situations in which it is necessary to integrate UNIX and PC environments.

4. References

Clark, SP, Evans GA, and Garner HR (1994). Informatics and Automation Used in Physical Mapping of the Genome. In "Biocomputing: Informatics and Genome Projects", Douglas Smith Ed., Academic Press, NY, pp. 13-49.

Dietrich W, Katz H, Lincoln SE, Shin H-S, Friedman J, Dracopoli NC, and Lander ES (1993). A Genetic Map of the Mouse Suitable for Typing Intraspecific Crosses. In "Genetic Maps, Locus Maps of Complex Genomes, Nonhuman Vertebrates", Vol 4, Stephen J. O'Brien, ed. Cold Spring Harbor Laboratory Press, Cold Spring Harbor, NY. pp. 110-142.

Green ED, Olson MV (1990). Systematic screening of yeast artificial-chromosome libraries by use of the polymerase chain reaction. Proc Natl Acad Sci US. 87:1213-1217

Goodman N, Rozen S, and Stein L (1993). Requirements for a Deductive Query Language in the MapBase Genome-Mapping Database. Workshop on Programming with Logic Databases, Vancouver, BC, Oct. 30, 1993.

Goodman N (1994). An Object Oriented DBMS War Story: Developing a Genome Mapping Database in C++. In "Modern Database Management: Object-Oriented and Multidatabase Technologies", Won Kim, Ed., ACM Press, NY.

Lander ES, Green P, Abrahamson J, Barlow A, Daly M., Lincoln S, Newburg L (1987). MAPMAKER: an interactive computer package for constructing primary genetic linkage maps of experimental and natural populations. Genomics 1: 174-181.

National Research Council, Committee on Mapping and Sequencing the Human Genome (1988) "Mapping and Sequencing the Human Genome", National Academy Press, Washington DC.

Kerlavage AR, Adams MD, Kelley JC, Dubnick M, Powell J, Shanmugam P, Venter JC, and Fields C (1993). Analysis and Management of Data from High_Throughput Sequence Tag Projects. in "Proceedings of the 26th Annual Hawaii International Conference on System Sciences", Trevor N. Mudge, Veljko Milutinovic and Lawrence Hunter, Eds. IEEE Computer Society Press. pp. 585-594.

Kernighan BW, and Plauger PJ (1976) "Software Tools", Addison-Wesley, NY, 1976.

Sulston J, Du Z, Thomas K, Wilson R, Hillier L, Staden R, Halloran N, Green P, Thierry-Mieg J, Qui L, Dear S, Coulson A, Craxton M, Durbin R, Berks M, Metzstein M, Hawkins T R, Ainscough R, and Waterston R (1992). The *C. elegans* genome sequencing project: A beginning. Nature 356:37-41.

Wall, L, and Schwartz RL (1991). "Programming perl", O'Reilly & Associates, Inc., Sebastopol, CA.

5. Biographies

Lincoln Stein is a MD/PhD pathologist and molecular biologist from Harvard with a persistent interest in software development including hypermedia-based medical education software, image analysis software, and desktop publishing tools. He is assistant group leader of the MIT Genome Center informatics core and takes the credit for much of the user interface design (or lack thereof) presented in this paper. **Andre Marquis** is a software engineer with a BA in Cognitive Science from the University of Rochester and is the implementor of the spreadsheet interface described in this paper. **Robert Dredge** is a software engineer with a BA in Biology from MIT; his contribution includes the automated error checking tools. **Mary Pat Reeve** is a software engineer with a BA in Biology from MIT, who is responsible for implementing much of the automated sequence analysis described in this paper. **Mark Daly** is a software engineer with a BA in Physics from MIT; his contribution includes the integration of MAPMAKER with MapBase. **Steve Rozen** is a PhD in Computer Science from New York University whose background includes the development of banking information systems on Wall Street. **Nathan Goodman** is a former professor of Computer Science at Harvard and former Chief Computer Scientist of Kendall Square Research Corp. He is the group leader of the MIT Genome Center informatics core and is responsible for the design and implementation of MapBase.

A Text Retrieval Package for the Unix Operating System

Liam R. E. Quin,

SoftQuad Inc. (lee@sq.com)

Abstract

This paper describes *lq-text*, an inverted index text retrieval package written by the author. Inverted index text retrieval provides a fast and effective way of searching large amounts of text. This is implemented by making an index to all of the natural-language words that occur in the text. The actual text remains unaltered in place, or, if desired, can be compressed or archived; the index allows rapid searching even if the data files have been altogether removed.

The design and implementation of *lq-text* are discussed, and performance measurements are given for comparison with other text searching programs such as *grep* and *agrep*. The functionality provided is compared briefly with other packages such as *glimpse* and *zbrowser*.

The *lq-text* package is available in source form, has been successfully integrated into a number of other systems and products, and is in use at over 100 sites.

1. Introduction

The main reason for developing *lq-text* was to provide an inexpensive (or free) package that could index and search fairly large corpora, and that would integrate well with other Unix tools.

Low Cost Solution: There are already a number of commercial text retrieval packages available for the Unix operating system. However, the prices for these packages range from Cdn\$30,000 to well over \$150,000. In addition, the packages are not always available on any given platform. A few packages were freely available for Unix at the time the project was started, but generally had severe limitations, as mentioned below.

A tool for searching large corpora: Some of the freely available tools used *grep* to search the data. While *lq-text* is $O(n)$ on the number of matches, irrespective of the size of the data, *grep* is $O(n)$ on the size of the data, irrespective of the number of

matches. This limits the maximum speed of the system, and searching for an unusual term in a large database of several gigabytes would be infeasible with *grep*. Other packages had limits such as 32767 files indexed or 65535 bytes per file. Tools available on, or ported from, MS/DOS were particularly likely to suffer from this malaise.

Unix-based: Unix has always been a productive text processing environment, and one would want to be able to use any new text processing tools in combination with other tools in that environment.

1.1 Results and Benefits

Timings are given in detail below, along with a perspective on how the above goals were met. As a brief example, a search over the SunOS 4.1.1 manual pages for “\<core dump\>” on a Sun 4/110 took 52 seconds with *grep*, and 1.3 seconds with *lq-text*. In addition, *lq-text* automatically finds matches of a phrase even if there is a newline or punctuation in the text between two words. It is also possible to combine *lq-text* searches, finding documents containing all of a number of phrases. Complex queries can be built up incrementally.

2. Design Goals

The main goals of any text retrieval package were outlined briefly in the introduction. *lq-text* was developed with some more specific goals in mind, described in the following sections.

2.1 Limited Memory

Developed originally on a 4MByte 386 under 386/ix, *lq-text* does not assume that *any* of its index files will fit completely into memory. As a result, indexing performance does not degrade significantly if the data does not fit into main, or even virtual, memory.

2.2 Offline Storage

Once files are indexed, *lq-text* does not need to consult them again during the searching process. The

result of a query is a list of matching files, together with locations within those files. The original text files are needed if the user wants to view the matched documents, but it turns out that the file names and document titles are often sufficient.

2.3 Matching With Certainty

Packages that use hashing or probabilistic techniques often return results that *might* match the user's query. A 'bad drop scan' is then used to reject the false hits [Lesk78]. These techniques are incompatible with the Offline Storage requirement, since a bad drop scan may not be possible.

2.4 Accurate Phrase Matching

The package should be able to match a phrase word for word, including getting the words in the right order and coping with uneven whitespace. It should also be able to match capitalisation and punctuation at least approximately, so that the user can constrain a search on someone's name, *Brown* for example, to match *Brown* but not *brown* in the text. Only the first letter of each word is inspected for capitalisation, in order to minimise the data stored in the index. This is sufficient for most queries.

In the literature, the terms *recall* and *precision* are used to refer to the proportion of relevant matches retrieved and the proportion of retrieved matches that are relevant, respectively; the goal of *lq-text* is to have very high precision, and to give the user some control over the recall. These terms are defined more precisely in the literature [Clev66], [Salt88], and are not discussed further in this paper. The term *accuracy* is used loosely in this paper to refer to literal exactness of a match—for example, whether *Ankle* matches *ankles* as well as *ankle*. In an inaccurate system, *Ankle* might also match a totally dissimilar word such as *yellow*. The information that *lq-text* stores in the database enables it to achieve a high level of accuracy in searching for phrases.

2.5 Updatable Index

It should be possible to add new documents to the index at any time. It should also be possible to *unindex* documents, or to *update* documents in place, as well as to inform the package when documents are renamed or moved.

2.6 Unix toolkit approach

It should be possible to manipulate search results with standard Unix tools such as *awk* and *sed*. This must be done in a way consistent with the Offline Storage requirement. For example, the user should be able to see the first seven matches using *head -7* or *sed 7q* without having to search the documents themselves.

2.7 Summary

This paper shows how some of the goals have been met, and indicates where work is still in progress on other goals. All of the original design goals are still felt to be relevant. With continued improvements in price/performance ratios of disks, the offline storage goal may become less important, but the design philosophy is still very important for CD-ROM work.

3. Technology Overview

This section gives a brief background to some of the main approaches to text retrieval.

3.1 Signatures

Packages based on *signatures* keep a hash of each document or block of each document. The idea is to reduce I/O by identifying those blocks which *might* contain a given word. This method does not store enough information to match phrases precisely, and software relying on it needs to scan documents to eliminate bad drops [Salt88], [Falo85], [Falo87a], [Falo87b]. A widely distributed publicly available system, *zbrowser*, uses a cross between block signatures and the Document Vector method discussed below. This system can answer proximity-style queries (these two words in either order, near each other) fairly well, but does not handle searching for phrases [Zimm91].

3.2 Full Text Inverted Index

A *full text inverted index* consists of a record of every occurrence of every word, and hence is generally the largest in size of the indexes discussed; the index also allows the highest *accuracy* (but not necessarily highest precision, see Future Work below). The larger index increases I/O needed for searching, but on the other hand there is no need to scan documents for bad drops [Salt88], [Mead92].

3.3 Document Vector

This strategy keeps a record of every file in which each word appears; one could call it a *partial inverted index*. This is usually much smaller than a full text inverted index, but cannot be used to find phrases directly without a bad drop scan. A recent example is *Glimpse*; this and other examples are mentioned in [Salt88], [Mead92], and [Orac92]. *Glimpse* also appears to be restricted to searching a single file system at a time on a local machine.

3.4 Relational Tables

One way of implementing a full or partial inverted index is by storing each occurrence of each word in a cell of a relational table. This is generally by far the least efficient of the strategies discussed, with indexes typically three times the size of the data, but it is also the easiest to implement robustly.

3.5 DFA or Patricia Tree

These systems store a data structure representing a deterministic finite state automaton that, when executed against the query, will reach an ‘accept’ state representing all matches. They are usually byte rather than word oriented, although they can be written either way. It is difficult to allow updates to such an index, and the algorithms are fairly complex. Knuth describes Patricia trees in some detail [Knut81]; a sample in-memory implementation, *Cecilia*, was described by Tsuchiya [Tsuc91]; PAT, the Oxford English Dictionary (OED) software, also uses Patricia trees [Bray89], [Fawc89].

3.6 Other

Many other approaches are possible. There are several schemes that actually replace the original data with, for example, a multiply-threaded linked list, so that the data can be recreated from the index. This has an unfortunate and well-known failure mode, in which the reconstituted text uses incorrect words. Other schemes include sub-word indexing, either on individual bytes or on *n-grams*, although these usually fall into the ‘DFA’ category above in terms of their characteristics.

4. The *lq-text* Design

A full text inverted index was chosen to meet the design goals. In particular, this is the only strategy which allows accurate matching of phrases without reverting to a bad drop scan.

In order to make the index smaller, however, the list of matches for each word is compressed, as described in detail in the Implementation section below.

The package is implemented as a C API in a number of separate libraries, which are in turn used by a number of separate client programs. The programs are typically combined in a pipeline, much in the manner of the probabilistic inverted index used by *refer* and *hunt* [Lesk78].

The *lq-text* package includes a set of *input filters* for reading documents into a canonical form suitable for the indexing program *lqaddfile*, to process; a set of search programs; and programs that take search results and deliver the corresponding text. There are also wrappers so that users don’t have to remember all the individual programs.

5. The *lq-text* Implementation

This section describes the implementation of *lq-text*.

5.1 Information Stored

Information is stored about the documents indexed, and about each distinct *word-form* (for example, information about occurrences of *sock* and *socks* is all stored under *sock*; see the discussion of stemming

under *Indexing* below).

Three distinct kinds of index are used. The first uses *ndbm*, a dynamic hashing package [Knut81]. The second type of index is a fixed record size random-access file, read using *lseek(2)* and a cache. Finally, the third index contains linked lists of variable-sized blocks; the location of the head of each chain is stored in one of the other indexes, as detailed below. This third index is used to store the variable-sized data constituting the list of matches for each word-form. The combination of the three index schemes allows fast access and helps to minimise space overhead.

Dynamic Hashing Packages: These have a number of desirable properties, including *minimal disk access*, since usually only two disk accesses are needed to retrieve any item, and *automatic expansion*, since the hash function simply gets wider as the database grows, allowing updates at any time. In addition, the technology is widely available, since many Unix systems include *ndbm*, and there are also much faster *ndbm*-clone implementations available.

Two *ndbm*-clone packages are distributed with *lq-text*. One of these, *sdbm*, has been widely distributed and is very portable [Yigi89]. The other, *db*, is part of the 4.4BSD work at Berkeley, and is described in [Selt91]. A general discussion on implementing such packages was distributed in [Tore87]. See also *ndbm(1)*.

Two *ndbm* databases are used: one maps file names into File Identifiers (*FID*), and the other maps natural-language words into Word IDentifiers (*WID*). **Fixed Record Indexes:** A word identifier (*WID*) as obtained from the *ndbm* word map is taken as a record number into a fixed size record file, *widindex*. The information found in this record is described later; for now, it suffices that one of the fields is a pointer into the linked lists of blocks in the final file, *data*.

Linked Lists of Blocks: Each word can occur many times in any number of files. Hence, a variable-sized data structure is needed. A linked list of 64-byte disk blocks is used. However, where adjacent blocks in the data correspond to the same thread, they are coalesced, rather as in the Berkeley Fast File System [McKu83]. Although an *lseek(2)* and a *read(2)* may be required for each 64-byte block, the Unix block buffer cache makes this arrangement relatively efficient. A Least Recently Used (LRU) cache holds a number of 16 Kbyte segments of these 64-byte blocks, giving a significant speedup, especially over NFS, where *write(2)* is synchronous.

5.2 Per-file Information

Each document is assigned a File Identifier when it is indexed. A File Identifier, or *FID*, is simply a number.

Conceptually, this number is then used to store and retrieve the information shown in Table 1.

Field	Example
location	/home/hermatach/cabochon/letters
name	tammuz.ar(june-14.Z)
title	<i>Star Eyes watched the jellycrusts peel</i>
size	1352 bytes
last indexed	12th Dec. 1991

Table 1: Per-file Information

In the table, /home/hermatach/cabochon/letters is an absolute path to a directory; if a relative or unrooted path is given, *lq-text* will search along a document path specified in the database configuration file, or by the DOCPATH environment variable. The search is performed on retrieval, so that the prefix to the path needn't be stored in the database. The document name is here given separately to emphasise that once a document has been indexed, it can be moved, or even compressed and then stored as a member of an archive, as here: *lq-text* will automatically extract june-14.Z from the ar-format archive tammuz.ar and run *uncompress* on the result to retrieve the desired document.

The size and date fields shown in the table are used to prevent duplicate indexes of the same file, so that one can run *lqaddfile* repeatedly on all the files in a directory and add only the new files to the index (no attempt is made to detect duplicated files with differing names). In addition, the file size allows *lq-text* to make some optimisations to reduce the size of the index, such as reserving numerically smaller file identifiers for large files. The reasoning is that larger file are likely to contain more, and more varied, words. Since the file identifier has to be stored along with each occurrence of each word in the index, and since (as will be shown) *lq-text* works more efficiently with smaller numbers, this can be a significant improvement.

In fact, all of the above information except the document title is stored directly in the *ndbm* index. Using a multi-way trie would probably save space for the file locations, but the file index is rarely larger than 10% of the total index size. The DOC PATH configuration parameter and Unix environment variable supported by *lq-text* allow the file names to be stored as relative paths, which can save almost as much space as a trie would, and at the same time allows the user to move entire hierarchies of files after an index has been created. The document title is kept in a separate text file, since users are likely to want to update these independently of the main text.

5.3 Per-word Information

For each unique word (that is, for each *lemma*), *lq-text* stores the information shown in Table 2.

Field	Bytes
Word length	1
the Word itself	(<i>wordlength</i>)
Overflow Offset	4
No. of Occurrences	4
Flags	1
(Total)	10 + <i>word-length</i>

Table 2: Per-word Information

The word length and the word are stored only if the WordList feature is not disabled in the database configuration file. The Overflow Offset is the position in the data overflow file (described below) at which the data for this word begins. The remainder of the word index record is used to store the first few matches for this word. In many cases, it turns out that all of the matches fit in the word index, and the Overflow Offset is set to zero.

The flags are intended to be a bitwise 'and' of all the flags in the per-word entries described below, but this is not currently implemented, and the space is not reserved. When implemented, this will let *lq-text* detect whether all occurrences of a word have a given property, such as starting with a capital letter. This information can then be used to recreate the correct word form in generating reports, and when searching the vocabulary.

5.4 Per-occurrence Information

For each occurrence of each lemma (that is, for each *match*), the information shown in Table 3 is stored.

Field	Size in Bytes
FID (file identifier)	4
Block Number	4
Word In Block	1
Flags	1
Separation	1
(Total)	11

Table 3: Per-occurrence Information

Since, on average, English words are approximately four characters long, and allowing one character for a space between words, one might expect the index to be approximately double the size of the data from the per-match information alone. Many commercial text retrieval systems do as badly, or worse [Litt94].

In fact, the information is stored compressed. The matches are stored sorted first by FID and then by block within file, then by word in block. As a result, for all but the first match it is only necessary to store the difference from the previous FID. Furthermore, matches for a single FID are grouped together, so that it isn't necessary to repeat the FID each time. The information ends up being stored as follows:

Δ FID (difference from previous File IDentifier)
Number of following matches using this FID
Block In File
Word In Block
Flags (if present)
Separation (if not 1)
Δ Block In File
Word In Block
Flags (if different from previous)
Separation (if not 1)

Storing lower-valued numbers makes the use of a variable-byte representation particularly attractive. The representation used in *lq-text* is that the top bit is set on all but the last byte in a sequence representing a number. Another common representation is to mark the first byte with the number of bytes in the number, rather like the Class field of an Internet address, but this means that fewer bits are stored in the first byte, so that there are many more two-byte numbers.

The flags are stored only if they are different from the previous match's flags. This is indicated on the disk by setting the least significant bit in the Word In Block value; this bit is automatically bit-shifted away on retrieval. Further, the separation is only stored if the flags indicate that the value is greater than one (whether or not the flags were stored explicitly). The combination of delta-coding, variable-byte representation and optional fields reduces the size of the average match stored on disk from eleven to approximately three bytes. For large databases, the index size is about half the size of the data. Furthermore, since *lq-text* has enough information stored that it can match phrases accurately without looking at the original documents, it is reasonable to compress and archive (in the sense of *ar(1)*) the original files. When presenting results to the user, *lq-text* will fetch the files from such archives automatically.

6. Programs and Algorithms

This section describes the programs used first to create and update *lq-text* indexes, and then to retrieve data, and outlines the main algorithms used.

6.1 Indexing

Storing the Per-File information: Each input file is read a word at a time. A word is defined to be a Word-Start character followed by zero or more WithinWord characters. In addition, a character of type Only-WithinWord may occur any number of times within a word, but if it occurs two or more times in a row, the first occurrence is taken as the last character in the word. This allows for the apostrophe in possessives (*the James'*) as well as in such words as *can't*. Words shorter than MinWordLength are rejected, and the next word read successfully will have the WPF_LASTHAD-LETTERS flag set. Words longer than MaxWordLength are truncated to that length. In addition, if any punctuation was skipped in looking for the start of the word, the WPF_UPPERCASE flag is set on this word.

The word is then looked up to see if it was in the common words file. If so, it is rejected and the next successfully read word will have the WPF_LAST-WASCOMMON flag set. In addition, whenever the start of this word is more than one character beyond the start of the previous successfully read word—as in the case that a common word, or extra space or punctuation, was skipped—the next successfully read word will have the WPF_HASSUFFBEFORE flag set, and the distance will be stored in the Separation byte shown in Table 3.

Common word look-up uses linear search in one of two sorted lists, depending on the first letter of the word. Using two lists doubled the speed of the code with very little change, but if more than 50 or so words are used, the common word search becomes a significant overhead; a future version of *lq-text* will address this.

After being accepted, the word is passed to the *stemmer*. Currently, the default compiled-in stemmer attempts to detect possessive forms, and to reduce plurals to the singular. For example, *feet* is stored as a match for *foot*, rather than in its own entry; there will be no entry for *feet*. When the stemmer decides a word was possessive, it removes the trailing apostrophe or 's, and sets the WPF_POSSESSIVE flag. When a plural is reduced to its singular, the WPF_WASPLURAL flag is set.

Other stemming strategies are possible. The most widespread is Porter's Algorithm, and this is discussed with examples in the references [Salt88], [Frak92]. Such an alternate stemmer can be specified in the configuration file. Porter's algorithm will conflate more terms, so that there are many fewer separate index entries. Since, however, the algorithm does not attempt etymological analysis, these conflations are often surprising.

The converted words are then mapped to a WID

as described above, and stored in an in-memory symbol table. Whenever the symbol table is full, and also at the end of an index run, the pending entries are added to the index, appending to the linked-list chains in the data file. The ability to add to the data for an individual word at any time means that an *lq-text* index can be added to at any time.

Compression: As mentioned above, numbers written to the index are stored in a variable-byte representation. In addition, the numbers stored are the difference between the current and previous values in a sequence.

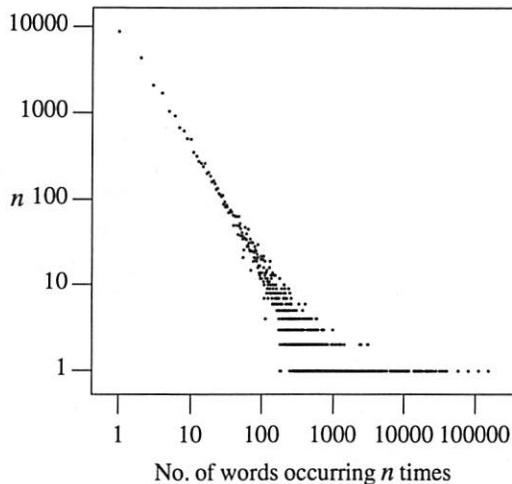


Figure 1: Vocabulary Distribution

Figure 1 shows the vocabulary distribution graph for the combined index to the King James Bible, the New International Version, and the complete works of Shakespeare, a total of some 15 megabytes of text (the NIV is in modern English; the other two are in 16th-century English). It can be seen from the graph that a few words account for almost all of the data, and almost all words occur fewer than ten times. The frequency f of the n th most frequent word is usually given by *Zipf's Law*:

$$f = k(n + m)^s \quad \dots \quad \dots \quad \dots \quad \dots \quad [1]$$

where k , m and s are nearly constant for a given collection of documents [Zipf49], [Mand53]. As a result, the optimisation whereby *lq-text* packs the first half dozen or so matches into the end of the fixed-size record for that word, filling the space reserved for storing long words, is a significant saving. On the other hand, the delta encoding gives spectacular savings for those few very frequent words: 'the' occurs over 50,000 times in the SunOS manual pages, for example; the delta coding and the compressed numbers reduce the storage requirements from 11 to just over

three bytes, a saving of almost 400,000 bytes in the index. Although Zipf's Law is widely quoted in the literature, the author is not aware of any text retrieval packages that are described as optimising for it in this way.

6.2 Retrieval

This section describes the various programs used to retrieve information from the index, and some of the algorithms and data structures used.

Simple Information: The *lqfile* program can list information about files in the index. More usefully, *lqword* can list information about words. For example, the script

```
lqword -A | awk '{print $6}' | sort -n
```

was sufficient to generate data for a *grap(1)* plot of word frequencies shown above (see Figure 1).

It is also possible to pipe the output of *lqword* into the retrieval programs discussed below in order to see every occurrence of a given word.

Processing a Query: A query is currently a string containing a single phrase. The database is searched for all occurrences of that phrase. In order to process a query, a client program first calls *LQT_StringToPhrase()* to parse the query. This routine uses the same mechanisms to read words from the string as the indexer (*lqaddfile*), and the same stemming is performed.

The client then calls *LQT_MakeMatches()*, which uses the data structure to return a set of matches. A better than linear time algorithm, no worse than $O(\text{total number of matches})$ is used; this is outlined in Algorithm 1, and the data structure used is illustrated in Figure 2.

This appears $O(m^w)$ at first sight, if there are w words in the phrase each with w matches, but the search at [3] resumes at the 'current' pointer, the high-water mark reached for the previous word at [1]. As a result, each match is inspected no more than once. For a phrase of three or more words, the matches for the last word of the phrase are inspected only if there is at least one two-word prefix of the phrase. As a consequence, the algorithm performs in better than linear time with respect to the total number of matches of all of the words in the phrase. In addition, although the data structure in the figure is shown with all of the information for each word already read from disk, the algorithm is actually somewhat lazy, and fetches only on demand.

Sample Program: The *lqphrase* program takes its arguments one at a time, treats each as a query, and processes it in turn as described. The main part of the source for *lqphrase* is shown in the Appendix, to illustrate this part of the C API.

```

[1] For each word in the first phrase {
    [2] for each subsequent word in the phrase {
        [3] is there a word that continues the match
            rightward? {
                Starting at current pointer, scan forward for a
                match in the same file;
                Continue, looking for a match in the same
                block, or in an adjacent block;
                Check that the flags are compatible
                If no match found, go back to [1]
                Else, if we're looking at the last word {
                    Accept the match
                } else {
                    continue at [2]
                }
            }
        }
    }
}

```

Algorithm 1: Phrase matching

6.3 Ranking of Results

A separate program, *lqranks*, combines sets of results and sorts them. Currently, only boolean ‘and’ and ‘or’ are available. Quorum ranking, where documents matching all of the phrases given are ranked above those that match all but *n* phrases, will be in the next release.

Statistical ranking—for example, where documents containing the given phrases many times rank more highly than those containing the phrases only once—is also planned work. See [Salt88]. Statistical ranking and document similarity, where a whole document is taken as a query, should also take document length into account, however; this is an active research area [Harm93].

The initial implementation of *lqranks* used *sed* and *fgrep*. This was improved by Tom Christiansen to use *perl*, and then coped with larger results sets (*fgrep* has a limit), but was slower.

The current version is written in C. For ease of use, *lqranks* can take phrases directly, as well as lists of matches and files containing lists of matches.

The algorithms in *lqranks* are not unusual; the interested reader is referred to the actual code.

6.4 Presentation of Search Results

The retrieval programs discussed so far—*lqword*, *lpphrase*, and *lqranks*—return an ASCII match format as follows:

- [1] Number of words in query phrase
- [2] Block within file
- [3] Word number within block

[4] File Number

[5] File Name and Location

(Items [4] and [5] are optional, but at least one must be present.)

In itself, this is often useful information, but the programs described below—*lqkwic* and *lqshow*—can return the corresponding parts of the actual documents.

Key Word In Context listings: The *lqkwic* program fetches the data from the original documents referred to by the given matches. It presents the results in the format used by a permuted, or ‘key word in context’, index. *lqkwic* has a built-in *little language* [Bent88] to control the formatting of the results, and uses lazy evaluation to maximise its efficiency. This program can be used to generate SGML concordances, for example, or even simply to expand the file name in each match into an absolute path. See the Appendix for sample *lqkwic* output.

Converting to line numbers: For use with editors and pagers such as *ex*, *nvi*, *less* and *more*, the *lqbyteline* program converts matches to (file, line-number) pairs. Unfortunately, although all of these editors and pagers understand a *+n file* option to open the named file at the given line number, the options applies only to the first file opened subsequent files are opened at the first line (*nvi* goes so far as to convert subsequent *+n* options to *-n* options, but then tries to edit a file of that name).

Text In Place: *lqshow* is a *curses*-based program to show part of a file, with the matched text highlighted.

6.5 Combined Interfaces

Two interfaces to *lq-text* are currently included in the distribution. These allow a combination of searching and browsing in a single interactive session.

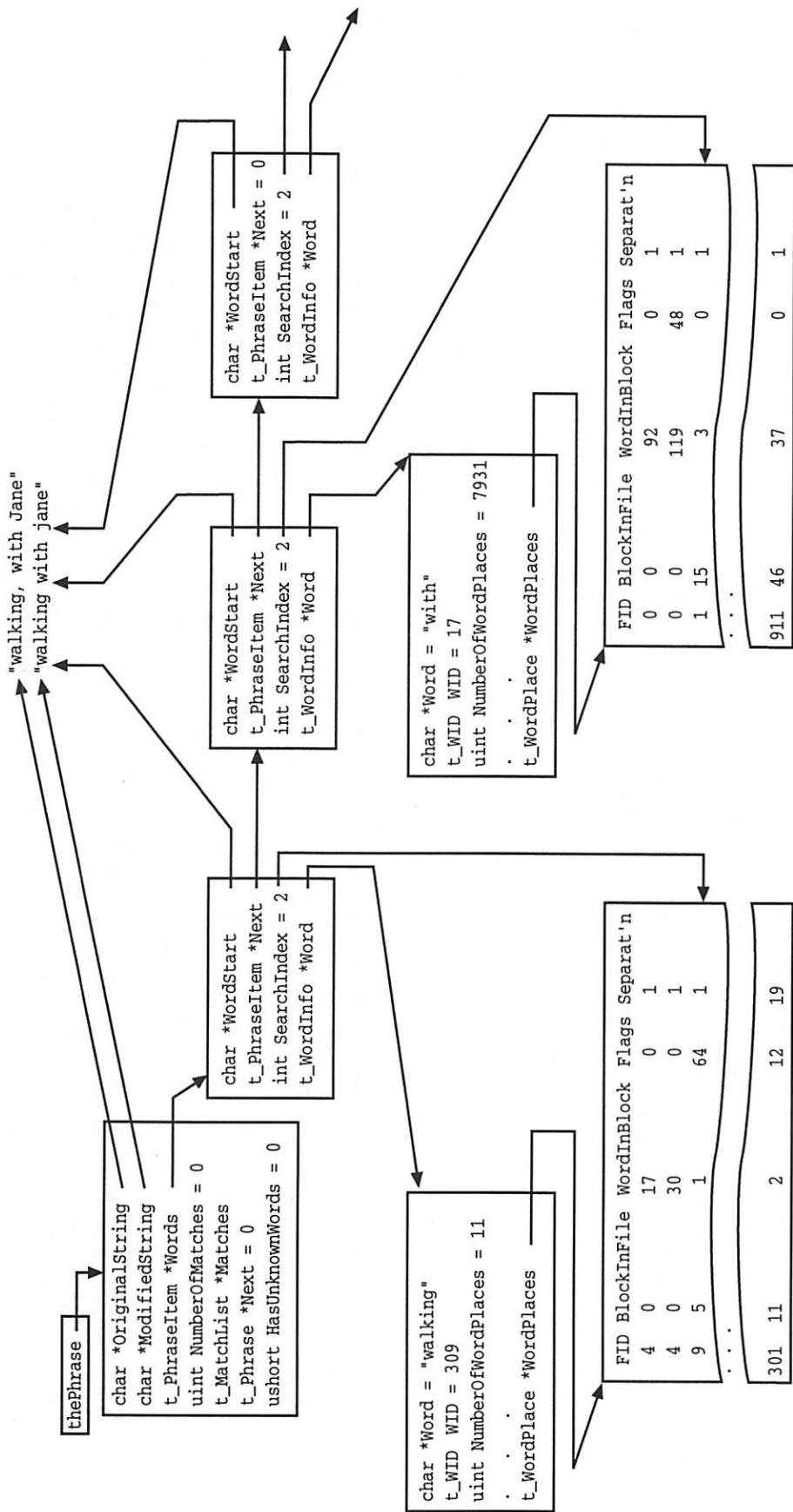
lqtext: This is a simple curses-based front end.

lq: This is a shell script that combines all of the *lq-text* programs with a simple command language. It requires the System V shell (with shell functions and the ability to cope with an 815-line shell script!).

The original purpose of *lq* was to demonstrate the use of the various *lq-text* programs, but *lq* is widely used in its own right. A brief example of using *lq* to generate a keyword in context listing from the netnews news.answers newsgroup is shown in the Appendix.

7. Performance

This section describes some work that was done to measure and improve the performance of *lq-text*, and then gives some actual measurements and timing comparisons with other systems.



Notes: The WordInfo for each word in the phrase also has its own WordPlace, with BlockInFile, WordInBlock, Flags and Separation set as if the query string was itself an indexed file. These values are interpreted more or less strictly depending on the current Phrase Match Precision Level.

The arrays of `t_WordPlace` structures shown here are usually very large. The `SearchIndex` pointers (actually integers) are indexes into the last row of each array that has been inspected for a match. This index allows the algorithm to resume each search where it left off, so that the complexity is no worse than linear.

Figure 2 – Data Structure used in Matching a Phrase (simplified)

7.1 Profiling

lq-text originally took over 8 hours to index the King James Bible on a 25 MHz under 386/ix. Extensive profiling, and careful tuning of cache algorithms, improved performance dramatically: the time to index the Bible has been reduced to under five minutes.

Function Calls: Although most C compilers have a fairly low function call overhead these days, it's still not trivial. Functions called for every character of the input were folded inline, and those called for every word were made into macros in many cases. Understanding *why* each function was called the number of time it was proved a big help both in speeding up the programs and in debugging them.

At one point, *lqaddfile* was spending over 40% of its time in *stat(2)*. It turned out that it was opening and closing an *ndbm* database for every word of input, which was suboptimal.

Now, most of the routines spend more than half of the time doing I/O, and no single function accounts for more than 10% of the total execution time.

7.2 Timings

The performance of *lq-text* is compared with SunOS 4.1 *grep*, GNU *grep* (*ggrep*), and Udi Manber's *agrep*. The *agrep* timings reflect only the simplest use of that program, since the goal was to generate comparable results. For *lq-text*, the time to build the index is also reported. Recall that the index only needs to be built once.

The following searches were timed; since the results for the various forms of *grep* were always very similar for any given set of files, the *grep* timings are only given once for each collection of data.

0. Not There: something not in the index at all, a nonsense word; the time was always 0.0 for *lq-text*, as reported by *time(1)*, irrespective of the size of the database. This timing is therefore omitted from the table.
1. Not Found: a phrase made up of words that do occur, but not in the order given (if 'gleeful' and 'boy' each occur, but 'gleeful boy' does not, 'gleeful boy' would be such a search).

2. Common: a phrase that occurs infrequently, but includes a relatively frequent word.

3. Unusual: a word or phrase that occurs infrequently

The following corpora were used:

Man Pages: The on-line manuals from SunOS 4.1, a total of twelve megabytes.

Bibles: The King James and New International bibles, and the Moby Complete Works of Shakespeare, a total of over 15 megabytes.

FAQ: The netnews news.answers newsgroup of approximately 1150 articles, totalling over 30 megabytes.

Timing Environment: A SPARCstation 10/30 (1 pro-

Index	egrep	ggrep	agrep	lq-text		
				[1]	[2]	[3]
man	41.5	37.1	45.9	1.8	0.0	0.3
Bible	60.2	55.5	67.7	1.8	1.2	0.1
FAQs	181.9	179.2	115.7	0.3	1.0	0.2

Table 4: Timings

Index	Size		Creation Time		
	Data	Index	real	user	sys
man	12M	6.5M	267.4	124.9	64.5
Bible	15M	6.6M	351.3	136.6	51.0
FAQs	41M	20.5M	1598.8	851.9	375.8

Table 5: Index Statistics

cessor, 30 MHz, no cache) with 64 MBytes of memory was used for the timings. The system was not equipped with Wide SCSI disks. The timings are given in real time, using *time(1)*, as this is the most important in practice. Each timing was performed several times, and an average taken of all but the first. This favours the *grep* algorithms somewhat, since it reduces the impact of the I/O that they do.

The *lq-text* timings do not include the time to produce the text of the match, for example with *lqkwic*. However, running *lqkwic* added less than one second of run-time for any except the very large queries, even when the data files were accessed over NFS.

The index overhead is approximately 50% of the size of the original data. This can be controlled to some extent using stop-words; the news.answers database used 79 stop-words, reducing the database by about 2 Megabytes. In addition, single-letter words were not indexed, although the presence of a single-letter word was stored in the per-word flags. The other databases used no stop words, and indexed words from 1 to 20 characters in length—the differences are because the FAQ index is accessed by a number of other users on our site.

Results: As one would expect, *lq-text* easily out-performs the *grep* family of tools. For queries producing a lot of matches, such as 'and the' (1790 occurrences in the SunOS manual pages), the time taken to print the matches dominates the run-time of *lqphrase*.

8. Ongoing and Future Work

This section describes speculative, planned and ongoing work.

8.1 The C API

The *lq-text* libraries (`liblqerror`, `liblqutil` and `liblqtext` itself) each provide a clear set of functions forming an Application Programmer's Interface (API). The process of tidying up the API is under way:

Documentation: The API is currently documented only by function prototypes in header files and by examples. Clearly this needs to change.

Completeness: The API isn't complete yet. For example, in `liblqerror` there's an `Eopen()` function which works like Unix `open(2)`, except that it provides error messages and can be made to exit on error. However, there is no `Eclose()` function yet.

Consistency: The structure of the API needs to be clear enough that one would be able to guess which library contains any given function; this is largely but not completely true now. Almost all functions have a prefix, such as `LQU_` in `LQT_ObtainWriteAccess()`, for example, for functions from `liblqtext`. A very few functions don't do this, and a few others are actually defined in client programs rather than in the library.

Configuration and Testing: Configuration is currently a case of editing a Makefile and a C header file, but several people have asked for something like the GNU auto-configuration package.

An *ad hoc* test suite is included with the *lq-text* distribution, but this needs to be made more formal, and to be run automatically when the software is built.

8.2 A User Interface

lq-text is primarily a text retrieval engine suitable for integration into other systems. However, experimental user interfaces have proved popular, and it is certainly expected that better interfaces will be provided in the future.

X11 interface: An X11 client based on the Fresco toolkit is planned, building on the work of Marc Chignel [Golo93], Ed Fox *et al.* [Fox93] and others. However, this work is awaiting the distribution of the Fresco toolkit with X11R6.

8.3 Functionality

In addition to the user interface, there are some specific features that are wanted:

Approximate matching: currently, *lq-text* can perform *egrep*-style matches against the vocabulary in the index; it would be interesting to extend this to *agrep*-style approximate patterns, and to integrate it into the main query language, so that

“core ^dump.*~/” might match ‘core dumped’, using approximate matching only for the second word in the phrase.

Complex queries: If it desired to support queries that

are themselves complex, or that refer to the structure of documents stored marked up in SGML format [Stan88], perhaps building on the work of Forbes Burkowski [Burk92]. Allowing a more complex syntax in a query has to be done carefully, so that the language is both straightforward and general. Handling structured documents also entails an extended query parser. At the same time, Fuzzy Logic [Zade78] and limited recognition of anaphoristic references is proceeding. It may also be possible to perform experiments in clustering, in the manner of some of the recent work at Xerox [Cutt93].

Performance: Although *lq-text* is already pretty fast at both retrieval and indexing, it could certainly be made faster. Experiments with *mmap(2)* and with alternate cache algorithms are ongoing.

Run-time configuration: New parameters will include user-defined stemming (perhaps using stemming algorithms described by W. Frakes in [Frak92]), and allowing a partial (document-vector) index.

9. Acknowledgements

Richard Parry helped with the original design, and walked through code. Mark Brader has contributed C advice. Mark Bremner contributed the code to uncompress documents on the fly. Mark Moraes helped with porting and installation issues. Henry Spencer, Geoff Collyer and Ian Darwin helped with the licence and copyright. Many, many people have sent feedback and bug fixes. Ozan Yigit, Bob Gray and Kate Hamilton helped with the paper.

10. Conclusions

The *lq-text* package is freely available, and hence clearly meets the criterion of low cost given at the start of the Introduction above. The largest database indexed by the author at the time of writing (March 1993) occupies about 100 Megabytes, and it remains to be determined how suitable *lq-text* is at indexing and searching larger bodies of text, which was the second main goal given in the Introduction. The package does provide fast retrieval, and meets all of the Design Goals given above except for the abilities to unindex and update documents. These last two features are expected in the summer of 1994.

The source for *lq-text* is available for anonymous ftp from `ftp.cs.toronto.edu` in `/pub`. Updates are announced on a mailing list, `lq-text-request@sq.com`.

11. References

[Bent88] Bentley, Jon, “Little Languages,” in *More Programming Pearls*, Addison-Wesley, 1988. A clearly-written rationale for the use of little (or embedded) languages. This column first appeared in Comm. ACM. in August 1986.

- [Bray89] Bray, Tim, *Lessons of the New Oxford English Dictionary Project*, pp. 137–199, Usenix, Winter, 1989.
- [Burk92] Burkowski, Forbes J., “An algebra for hierarchically organized text-dominated databases,” *Information Processing & Management*, **28**, p. 333, 3, 1992.
- [Clev66] Cleverdon, C. W., Mills, J., and Keen, E.M., *Factors Determining the Performance of Indexing Systems, Volume 1 – Design*, Aslib Cranfield Research Project, Cranfield, 1966.
- [Cutt93] Cutting, Douglas R., Karger, David R., and Pedersen, Jan O., “Constant Interaction-Time Scatter/Gather Browsing of Very Large Document Collections,” in *Proc. 16th ACM SIGIR*, pp. 126–131, 1993. One of a number of papers reporting work at Xerox Parc on information retrieval
- [Falo85] Faloutsos, Christos, “Access Methods for Text,” *Computing Surveys*, **17**, 1, pp. 49–74, March 1985. Compares text retrieval methods for office systems
- [Falo87a] Faloutsos, Christos and Christodoulakis, Stavros, “Optimal Signature Extraction and Information Loss,” *ACM Trans. on Database Systems*, **12**, 3, pp. 395–428, Sept. 1987.
- [Falo87b] Faloutsos, Christos and Christodoulakis, Stavros, “Description and Performance Analysis of Signature File Methods,” *ACM Trans. on Office Systems*, **5**, 3, July 1987. A good overview of signatures.
- [Fawc89] Fawcett, Heather, *PAT User’s Guide*, Open Text, 1989.
- [Fox93] Fox, Edward A., France, Robert K., Sahle, Eskinder, Daoud, Amjad, and Cutter, Ben, “Development of a Modern OPAC: From REVTOLC to MARIAN,” TR 93-06, Virginia Polytechnic Institute and State University, 1993. A client-server Online Public Access Catalogue for a library, using the NeXTStep GUI.
- [Frak92] Frakes, William B. and Baeza-Yates, Ricardo, *Information Retrieval: Data Structures and Algorithms*, Prentice-Hall, 1992. An excellent introduction to the issues in implementing information retrieval systems. Examples in C for Unix, available by ftp from ftp.vt.edu in pub/reuse/ir-code.
- [Golo93] Golovchinsky, G. and Chignell, M.H., “Queries-R-Links: Graphical Markup for Text Navigation,” in *Proceedings of INTERCHI ’93, Amsterdam*, pp. 454–460, ACM Press., N.Y., April 1993. Presents a conceptually simple way for users to add and subtract terms from text retrieval queries, and raises issues about the trade-offs between pre-determined hypertext links and live text retrieval queries.
- [Harm93] Harman, Donna, “Overview of the First TREC Conference,” *Annual ACM SIGIR Conf.*, **16**, p. 36, 1993. At the SIGIR conference in 1993, some of the TREC participants reported that they had had difficulties using similarity techniques on long documents.
- [Knut81] Knuth, Donald, *The Art of Computer Programming, Vol III: Sorting and Searching*, Addison-Wesley, 1981.
- [Lesk78] Lesk, M. E., “Some Applications of Inverted Indexes on the Unix System,” in *V7 Unix Programmers’ Manual, Vol 2A*, Bell Laboratories, 1978.
- [Litt94] Littman, Dan, “AppleSearch 1.0,” *Macworld*, May 1994. A review of Apple’s ‘easy to administer, easy to use’ text retrieval software. Mentions that ‘the indexing process requires more than double the disk space of the original documents’.
- [Mand53] Mandelbrot, Benoit, “An informational theory of the statistical structure of language,” in *Communication Theory*, ed. Willis Jackson, pp. 486–502, Butterworths, 1953.
- [McKu83] McKusick, Marshall Kirk, Joy, William N., Leffler, Samuel J., and Fabry, Robert S., “A Fast File System for Unix,” CSRG Technical Report 83-147, 1983.
- [Mead92] Meadow, Charles T., *Text Information Retrieval Systems*, Academic Press, Toronto, 1992. Gives clear descriptions of full-text retrieval data structures and algorithms, although with a bias towards indexing only abstracts of books or of library catalogue entries.
- [Orac92] Oracle Corporation,, *SQL*TextRetrieval Version 2 Technical Overview*, 1992.
- [Salt88] Salton, Gerald, *Automatic Text Processing*, Addison-Wesley, 1988.
- [Selt91] Seltzer, Margo and Yigit, Ozan, *A New Hashing Package for Unix*, Usenix ’91, Dallas, TX, 1991.
- [Stan88] (ISO), International Organization for Standardization, “Information Processing—Text and office systems—Standard Generalized Markup Language (SGML),” ISO8879, 1988.
- [Tore87] Torek, Chris, *Re: dbm.a and ndbm.a archives*, net-news comp.unix newsgroup, 1987.
- [Tsuc91] Tsuchiya, Paul F., *A Search Algorithm for Table Entries with Non-contiguous Wildcarding*, Bellcore, 1991. Unpublished(?) description of Cecilia, a package using in-memory Patricia trees with efficient update and deletion.
- [Yigi89] Yigit, Ozan, *How to roll your own dbm/ndbm*, Unpublished Manuscript, 1989.
- [Zade78] Zadeh, L. A., “PRIF—a meaning representation language for natural languages,” *Int. J. Man-Machine Studies*, **10**, pp. 395–460, 1978. One of many of L. A. Zadeh’s papers arguing for modeling the ‘pervasive imprecision of natural languages’ (p.396).
- [Zimm91] Zimmerman, Mark, *Zbrowsr implementation*, 1991. Article in *para* mailing list, unpublished.
- [Zipf49] Zipf, George K., *Human Behavior and the Principle of Least Effort*, Addison-Wesley, Cambridge, MA., USA, 1949.

12. Appendix 1—Source for *lqphrase*

```
PRIVATE void
MatchOnePhrase(Phrase)
    char *Phrase;
{
    t_Phrase *P;

    if (!Phrase || !*Phrase) {
        /* ignore an empty phrase */
        return;
    }

    if ((P = LQT_StringToPhrase(Phrase)) == (t_Phrase *) 0) {
        /* not empty, but contained no plausible words */
        return;
    }

    /* PrintAndAcceptOneMatch() is a function that prints
     * a single match. It is called for each match as soon as it is
     * read from the disk. This means that results start appearing
     * immediately, a huge benefit in a pipeline.
    */
    if (LQT_MakeMatchesWhere(P, PrintAndAcceptOneMatch) <= 0L) {
        return;
    }
}
```

13. Appendix 2—Sample *lq* session

This listing shows part of a session using *lq*, a shell-script that uses *lq-text*. The *faq* command invokes *lq* after setting up environment variables and options to use the news.answers database.

```
: sqrex!lee; faq
Using database in /usr/spool/news/faq.db...
| Type words or phrases to find, one per line, followed by a blank line.
| Use control-D to quit. Type ? for more information.
> text retrieval
>
Computer Science Technical Report Archive Sites == news/answers/17480
  1:v.edu> comments: research reports on text retrieval and OCR orgcode: ISRI
Interleaf FAQ -- Frequently Asked Questions for comp.text.interleaf == news/answers/17607
  2:g with hypertext navigation and full-text retrieval. 1.2. What platform
alt.cd-rom FAQ == news/answers/17643
  3:and Mac. 56. Where can I find a full text retrieval engine for a CDROM I a
  4:===== 56. Where can I find a full text retrieval engine for a CDROM I a
  5: I am making? Here is a list of Full-Text Retrieval Engines from the CD-PU
OPEN LOOK GUI FAQ 02/04: Sun OpenWindows DeskSet Questions == news/answers/18078
  6:OOK/XView/mf-fonts FAQs;lq-text unix text retrieval who is my neighbour?
OPEN LOOK GUI FAQ 04/04: List of programs with an OPEN LOOK UI == news/answers/18079
  7: Description: Networked, distributed text-retrieval system. OLIT-based fr
  8:OOK/XView/mf-fonts FAQs;lq-text unix text retrieval who is my neighbour?
[comp.text.tex] Metafont: All fonts available in .mf format == news/answers/18080
  9:OOK/XView/mf-fonts FAQs;lq-text unix text retrieval who is my neighbour?
OPEN LOOK GUI FAQ 03/04: the XView Toolkit == news/answers/18082
  10:OOK/XView/mf-fonts FAQs;lq-text unix text retrieval who is my neighbour?
Catalog of free database systems == news/answers/18236
  11:----- name: Liam Quin's text retrieval package (lq-text) vers
  12: are bugs. description: lq-text is a text retrieval package. That means
| Type words or phrases to find, one per line, followed by a blank line.
| Use control-D to quit. Type ? for more information.
> :less 9 (brings up the 9th match in a pager)
> :help
| Commands
|
| :help
```

```

Shows you this message.
:view [n,n1-n2,n...]
  Use :view to see the text surrounding matches.
  The number n is from the left-hand edge of the index;
  :set maxhits explains ranges in more detail.
:page [n,n1-n2,n...]
  Uses less -c (which you can set
  in the $PAGER Unix environment variable) to show the files matched.
:less [n,n1-n2,n...]
  The same as :page except that it starts on the line number
  containing the match, firing up the pager separately on
  each file.
:index [n,n1-n2,n...]
  This shows the index for the phrases you've typed.
:files [n,n1-n2,n...]
  This simply lists all of the files that were matched,
  in ranked order.
:prefix prefix-string
  Shows all of the words in the database that begin with that prefix,
  together with the number of times they occur.
:grep egrep-pattern>
  Shows all of the words in the database that match that egrep pattern,
  together with the number of times they occur (not the number of files
  in which they appear, though).
:set option ...
  Type :set to see more information about setting options.
:shell [command ...]
  Use /home/sqrex/lee/bin/sun4os4/msh to run commands.

Commands that take a list of matches (n,n1-n2,n...)
only work when you have generated a list of matches. If you don't give
any arguments, you get the whole list.

> :set
:set match [precise|heuristic|rough]
  to set the phrase matching level,
  currently heuristic matching (the default)
:set index
  to see a Key Word In Context (KWIC) index of the matches
:set text
  to browse the actual text that was matched
:set database directory
  to use the database in the named directory.
  Current value: /usr/spool/news/faq.db
:set rank [all|most|any]
  all presents only documents containing all the phrases you typed;
  most shows those first, and then documents with all but one of
  the phrases, and so on.
  any doesn't bother sorting the matches; this is the default,
  because it's the fastest.
  Currently: find documents containing all of the phrases
:set tags string
  determine whether SGML tags are shown or hidden in the KWIC index
:set tagchar string
  set the character used as a replacement for hidden SGML tags
:set maxhits n|all
  show only the first n matches (currently 200)
:set prompt string
  set the prompt for typing in phrases to string
> ^C
Interrupted, bye.
: sqrex!lee;

```


Probing TCP Implementations

Douglas E. Comer and John C. Lin *

Department of Computer Sciences

Purdue University

West Lafayette, Indiana 47907

Abstract

In this paper, we demonstrate a technique called *active probing* used to study TCP implementations. Active probing treats a TCP implementation as a *black box*, and uses a set of procedures to probe the black box. By studying the way TCP responds to the probes, one can deduce several characteristics of the implementation. The technique is particularly useful if TCP source code is unavailable.

To demonstrate the technique, the paper shows example probe procedures that examine three aspects of TCP. The results are informative: they reveal implementation flaws, protocol violations, and the details of design decisions in five vendor-supported TCP implementations. The results of our experiment suggest that active probing can be used to test TCP implementations.

1 Introduction

The Transmission Control Protocol (TCP) is a connection-oriented, flow-controlled, end-to-end transport protocol that provides reliable transfer and ordered delivery of data [14]. TCP is designed to operate successfully over communication paths that are inherently unreliable (i.e., they can lose, damage, duplicate, and reorder packets). The ability of TCP to adapt to networks of various characteristics and computer systems of various processing power makes TCP an important component in the fast expansion of the global Internet.

The original definition of TCP appears in RFC-793 [14]. Many researchers [2, 7, 8, 9, 11, 12, 18, 19] have identified problems and weakness of the protocol, and proposed solutions. RFC-1122 [1] updates and supplements the definition; to meet the TCP standard, an implementation must follow both RFC-793 and RFC-1122.

Although RFCs 793 and 1122 give a detailed description of TCP implementation, two TCP implementations that conform to the specifications can differ slightly because an implementor has some freedom to choose a software design, parameters, and to interpret the protocol standards. Although it is possible to deduce design decisions and parameters choices from the source code, understanding the operation of a complex software module like TCP can be difficult. In this paper, we demonstrate a technique called *active probing* used to study TCP implementations. Active probing is especially useful when source code is unavailable. Furthermore, it shows how the TCP code operates in the presence of other system components.

Active probing treats a TCP implementation as a *black box* and uses a set of procedures to probe the black box. By studying the way TCP responds to the probes, one can deduce characteristics of the implementation. The information that can be deduced depends on the probing procedures used. In this paper, we show three example procedures that examine three aspects of TCP. The results are informative: they reveal implementation flaws, protocol violations, and the details of design decisions in commercially available TCP implementations. The results of the experiment suggest that active probing can also be used to test TCP implementations.

Active probing operates much like traditional TCP trace analysis. It uses a software tool to capture TCP segments directed toward a particular TCP implementation as well as segments the TCP implementation sends in response. It then analyzes the trace data to find patterns that reveal characteristics of the TCP implementation. Unlike trace analysis, however, active probing uses specially designed probing procedures to induce TCP traffic instead of passively monitoring normal traffic on the network.

The software tools used to capture TCP segments and to assist in the analysis of the trace data are widely available, both in public domain and in commercial

*This work was supported in part by a fellowship from UniForum.

domain. RFC-1470 [4] gives a detailed catalog of such tools. All experiments reported in this paper use the tools from NetMetrix [6] to capture the TCP segments and to assist in the analysis of the trace data; we also wrote C programs to parse and analyze the captured data.

The experiments reported in this paper examine commercially available TCP implementations: Solaris 2.1, SunOS 4.1.1, SunOS 4.0.3, HP-UX 9.0, and IRIX 5.1.1. We chose these implementations because they are widely available in workstation operating systems. We only have the access to the source code of SunOS 4.0.3 and SunOS 4.1.1.

The remainder of this paper is organized as follows. Section 2 examines TCP retransmission time-out intervals for successive retransmission of a single data segment. Section 3 studies the keep-alive mechanism in some TCP implementations. Section 4 investigates TCP zero-window probing. Finally, section 5 draws conclusions and discusses future work.

2 Successive Retransmission Intervals In TCP

TCP uses an *acknowledgment and retransmission* scheme to ensure the reliable delivery of packets. When sending a packet, the sender starts a timer and expects an acknowledgment from the receiver within a *retransmission time-out (RTO)* period. If the sender does not receive an acknowledgment in that period, it assumes the packet was lost and retransmits the packet. The correct estimation of the retransmission time-out is vitally important to provide effective data transmission and avoid overwhelming the Internet by excessive retransmissions [11]. On one hand, if the sender uses a smaller RTO value than the actual packet round-trip time (RTT), unnecessary retransmissions occur. Moreover, if the packet round-trip time increase is due to network congestion, unnecessary retransmissions make the situation even worse and may lead to *congestion collapse* [12]. On the other hand, if the sender uses a larger RTO value, a lost packet causes the sender to wait longer than necessary, thus degrading throughput.

The calculation of the RTO value originally suggested in RFC-793 is now known to be inadequate and has been replaced. RFC-1122 specifies the new standard, which uses an algorithm described in Jacobson [8]. The new algorithm uses the measured RTT values to calculate a smoothed mean and a measure of the variance using a smoothed mean difference. The RTO is then calculated from the smoothed mean and

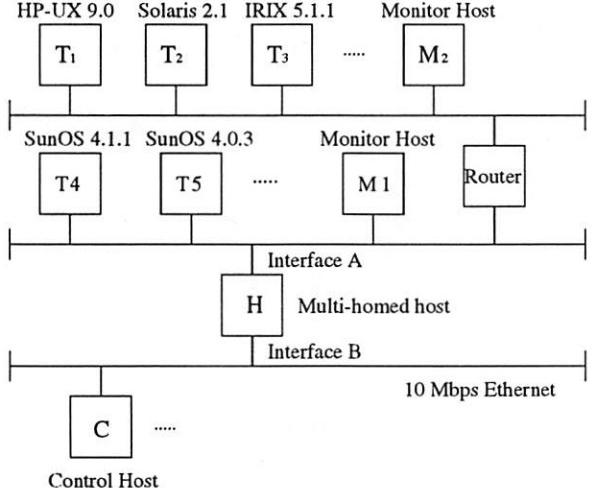


Figure 1: Configuration of networks and hosts to obtain successive retransmissions intervals in TCP

the variance. RFC-1122 specifies that TCP *must* implement this algorithm and *must* exponentially increase the RTO values for successive retransmissions of the same segment.

2.1 Probing Procedure

To determine how a TCP implementation chooses RTO values for successive retransmissions, we use the following probe procedure:

1. From a host to be tested, T , select a multi-homed host¹, H , as the destination (see Figure 1).
2. Let the IP address of one interface on H , say A , be the destination address that can be reached by T .
3. From T , open a TCP connection to the *discard* port [16] of host H via interface A , and start sending data.
4. Login to host H from a control host, C , via another interface, say B .
5. Disable interface A while the communication between host T and host H is in progress².

Disabling interface A while host T is sending data to the discard port of host H via interface A simulates a network failure between host T and host H , and it triggers retransmissions from T . Note that T runs the

¹A multi-homed host is a host that connects to more than two networks.

²We used the UNIX command `ifconfig` to disable the interface.

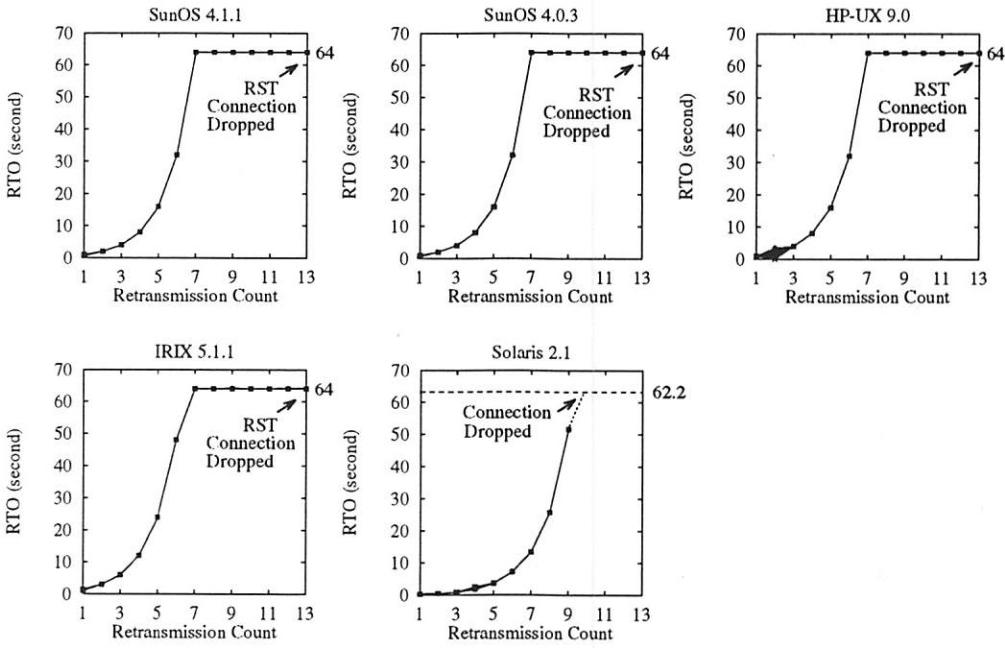


Figure 2: The TCP RTO intervals for successive retransmissions in a LAN environment

probe program; it is also the host on which TCP is being probed. To enable continued control of H while interface A is down, one must login to H from a control host (C) via interface B . C and interface B are connected to the same Ethernet.

Because the RTO estimate depends on the packet round trip time between a tested host and host H , all the TCP implementations tested run on hosts connected to 10 megabit per second (Mbps) Ethernets. The average load on the Ethernets during the experiment is less than 10% of capacity. The tested hosts are located at most one gateway from H (see Figure 1). The average round trip time of packets between a tested host and H during the experiments, measured using `ping`, is at most 10ms. To make the measurements more accurate, the monitor program that captures the TCP segments always runs on a host connected to the same Ethernet as the hosts being probed. (The monitor program runs on host M_1 or M_2 depending on which host is being probed.)

2.2 Results

For each TCP implementation, we conducted 30 experiments; Figure 2 shows the results. As the graphs in Figure 2 show, four of the probed operating systems, SunOS 4.1.1, SunOS 4.0.3, HP-UX 9.0, and IRIX 5.1.1, behave the same. Each increases the RTO val-

ues exponentially on successive retransmissions until it reaches a maximum RTO of 64 seconds. Each retransmits the same data segment twelve times; at the thirteenth transmission, each sends a reset (RST) segment (without data), drops the connection, and terminates the process that executes the probe program.

Solaris 2.1 TCP increases the RTO values for successive retransmissions and drops the connection after the ninth retransmission. The Solaris TCP does not send a RST segment after the ninth retransmission. However, it delays for 62.2 seconds³ before it drops the connection and terminates the process that executes the probe program.

RFC-1122 specifies a threshold, R_2 , for dealing with excessive retransmissions of the same segment by TCP. R_2 can be measured in units of time or as a count of re-

³Obtained by using $\sum_{i=1}^{30} (p_i - q_i)/30$, where p_i is the interval between the instance at which the probe program calls a `connect` routine to establish a connection and the instance at which the process that runs the probe program (called P) exits in the i -th experiment, and q_i is the interval between the instance at which TCP sends the first segment and the instance at which TCP sends the last segment as measured by the monitor program in the i -th experiment. The time interval, p_i , consists of three parts: α , q_i , and β , where α is the interval between the instance at which the probe program calls `connect` and the instance at which the first segment is sent, and β is the interval between the instance at which the last segment sent and the instance at which process P exits. Because α is small compared to β , $p_i - q_i$ is an approximation of β .

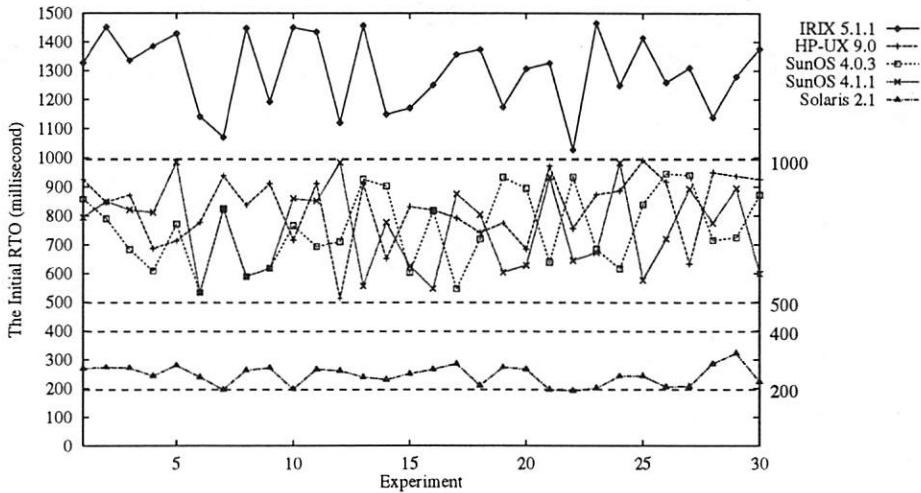


Figure 3: The initial RTO values in TCP implementations in a LAN environment

transmissions. When the number of retransmissions of the same segment reaches R2, TCP closes the connection. RFC-1122 specifies that R2 should correspond to at least 100 seconds. All the implementations probed meet the requirement. However, no implementation allows users to configure the value for R2 as RFC-1122 mandates.

The initial RTO values in TCP implementations are worth noting. In a local area network (LAN) environment that consists of 10 Mbps Ethernet segments with a average load of less than 10% of the available bandwidth, typical packet RTTs average less than 20 ms, and the variance (smoothed mean difference) of the packet RTTs averages less than 10 ms. So, a typical RTO value calculated from *mean plus variance* will remain under 100 ms. Figure 3 shows that the initial RTO values used by TCP implementations are all much higher than 100 ms. The large initial RTO values suggest that the implementations have imposed a lower bound on the RTO estimates.

2.3 The Lower Bound on RTO Estimates

There are two reasons for imposing a lower bound on the RTO estimates. First, the timer used to measure packet RTT may be too coarse for accurate measurements. For example, the 4.3BSD TCP (and most of its derivatives) uses a timer of 500 ms per tick to measure the packet round trip time and to schedule retransmissions [10]. In a LAN environment with typical packet RTT less than 20 ms, using such a timer to measure packet RTT accurately is impossible. Thus, a lower

bound filters out the RTT samples that are too small to measure accurately with a coarse granularity timer.

Second, imposing a lower bound on RTO estimates can improve throughput in a LAN environment. A LAN environment exhibits low packet loss and low average packet round trip time. Imagine a TCP implementation that uses a millisecond granularity timer to measure packet round trip time and to schedule retransmissions without imposing a lower bound on RTO estimates. Under normal load conditions, the smoothed RTT will be less than 10 ms and the variance (smoothed mean difference) is less than 5 ms. A sudden network delay or host processing delay that causes the RTT of a segment to exceed 20 ms⁴ will cause a retransmission of that segment even though the segment is not likely to be lost in transit. The redundant retransmission not only consumes network bandwidth and adds unnecessary processing overhead to the sender and receiver, but also forces the sender to a *slow start* mode [8] that reduces its transmission rate.

Another way of viewing the lower bound on the RTO estimates is to consider it a threshold for the RTO estimation algorithm to take effect. If the lower bound is set to infinity, TCP ignores the RTO estimates entirely (TCP makes no attempt to retransmit lost packets); if the lower bound is set to zero, TCP uses the RTO estimates for each transmission. Because the RTO estimation algorithm derives an estimate of future RTO from the previous RTT samples, it can only cover the fluctuations of packet RTT within a specific range. Any

⁴We calculate RTO as mean plus twice the variance.

Segment Number	Host A (Solaris 2.1)	Host H	Comment
992	S:2473 D:488 A:2002 W:9112 →		
993	S:3985 D:536 A:2002 W:9112 →		(ERROR! sequence # should be 2961)
994	← S:2002 D:0 A:2961 W:3608		
995	S:4521 D:448 A:2002 W:9112 →		
996	S:5009 D:536 A:2002 W:9112 →		
997	S:5545 D:448 A:2002 W:9112 →		
998	S:6003 D:536 A:2002 W:9112 →		
999	← S:2002 D:0 A:2961 W:4096		
1000	S:2961 D:536 A:2002 W:9112 →		(Transmission of the missing data)
1001	← S:2002 D:0 A:3497 W:4096		
1002	S:3497 D:488 A:2002 W:9112 →		(Transmission of the missing data)
1003	← S:2002 D:0 A:6569 W:4096		
....			

S: Sequence number, D: Number of data octets, A: Acknowledgment number, W: Window
Note: Only the last four digits of the sequence number and acknowledgment number are shown.

Figure 4: Illustration of an implementation flaw in Solaris 2.1 TCP.

sudden RTT fluctuations that exceed that range will trigger unnecessary retransmissions. On one hand, using a higher lower bound allows TCP to tolerate greater network delay fluctuations without triggering unnecessary retransmissions; but it makes TCP take longer to respond to lost packets. On the other hand, using a lower lower bound allows TCP to respond to lost packets quickly, but it may cause unnecessary retransmissions when network delay fluctuations exceed RTO estimations. Therefore, the lower bound on RTO estimates is a design parameter a TCP implementation must choose carefully.

As Figure 3 shows, the lower bound on the observed systems is a range of values⁵. IRIX 5.1.1 TCP has the largest lower bound (in the range of 1000 ms to 1500 ms) and Solaris TCP has the smallest lower bound (in the range of 200 ms to 400 ms). SunOS 4.1.1, HP-UX 9.0, and SunOS 4.0.3 has the lower bound set in the range of 500 ms to 1000 ms.

2.4 Implementation Flaw Found

In analyzing the probe results for Solaris 2.1 TCP, we have found an apparent implementation flaw. The symptom occurs in all 30 instances of TCP trace data we gathered. As Figure 4 illustrates, host A, running Solaris 2.1, sends data to the discard port of host H. Segment #992 has sequence number 2473 and carries 488 octets of data. The next data segment from A should have sequence number 2961 (2473+488).

⁵In reading the SunOS 4.1.1 and 4.0.3 TCP source code, we found that the inaccuracy in the timer algorithm for scheduling retransmissions can cause the lower bound on RTO to be a range of values.

Instead, segment #993 has sequence number 3985 (2473+488+1024). Apparently TCP has skipped 1024 octets in the sequence space! After 234 milliseconds, A transmits the missing 1024 octets of data in segment #1000 and segment #1002.

Note that the monitor program runs on host M_2 connected to the same Ethernet as A. Thus, the missing segments are not discarded by a gateway. Furthermore, the retransmissions of the missing data segments in segments #1000 and #1002 show that the error did not result from the monitor program missed the original transmissions. The same symptom also occurs in 10 of the 30 instances of the IRIX 5.1.1 trace data.

3 TCP Keep-alives

The TCP specification does not include a mechanism for probing idle connections. In theory, if a host crashes after establishing a connection to another host, the second machine will continue to hold the idle connection forever. Some TCP implementations include a mechanism that tests an idle connection and releases it if the remote host has crashed. Called TCP *keep-alive*, the mechanism periodically sends a probe segment to elicit response from the peer. If the peer responds to the probe by sending an ACK, the connection is *alive*. If the peer TCP fails to respond to probe segments for longer than a fixed threshold, the connection is declared *down* and the connection is closed.

According to RFC-1122, a TCP implementation *may* include the keep-alive mechanism. However, if TCP keep-alive is included, the applications *must* be able to turn it on or off in a per connection basis, and by

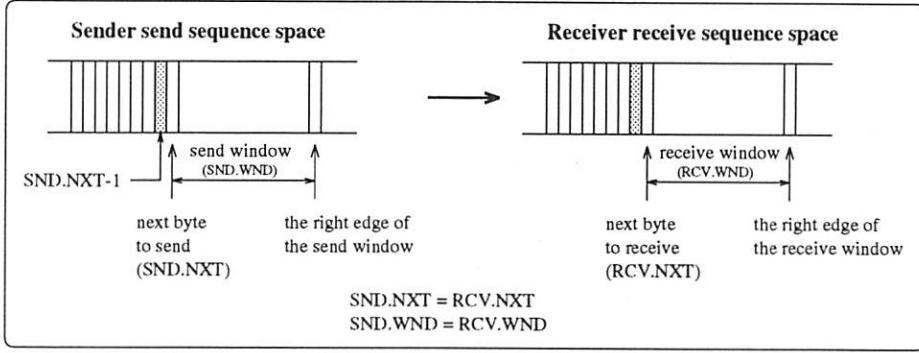


Figure 5: The sender’s send and receiver’s receive TCP sequence spaces when a connection is quiet

Operating System	Data size in Seg.	Sequence Number	ACK Seq. Number	Probing Interval
Solaris 2.1	N/A ⁸	N/A	N/A	N/A
SunOS 4.1.1	1 octect	SND.NXT-1	RCV.NXT-1	7200 sec.
SunOS 4.0.3	1 octect	SND.NXT-1	RCV.NXT-1	75 sec.
HP-UX 9.0	1 octect	SND.NXT-1	RCV.NXT-1	7200 sec.
IRIX 5.1.1	1 octect	SND.NXT-1	RCV.NXT-1	7200 sec.

Table 1: The results of TCP keep-alive probing in TCP implementations

default, it *must* be off. The threshold interval to send TCP keep-alives *must* be configurable and *must* default to 7,200 seconds (two hours) or more. Because TCP does *not* reliably transmit ACK segments that carry no data⁶, an ACK segment in response to the keep-alive probe may be lost. Therefore, a TCP should drop the connection only after a predefined number of keep-alive probes fail to elicit response from the peer.

3.1 Probe Procedure

We use the following probe procedure to study whether an implementation of TCP uses keep-alive, and, if so, how they implement it.

1. From a host to be tested, open a TCP connection to the discard port of another host.
2. Enable keep-alive on the connection.
3. Pause⁷ until a terminating signal occurs.

As Figure 5 illustrates, when a TCP connection is quiet, the sequence number of the sender’s next

⁶There is no retransmission timer set for an ACK segment that carries no data.

⁷C library function `pause()` may be used.

⁸No keep-alive segment observed in five observations; each observation lasted for 30 hours.

octet to send (SND.NXT) is the same as the sequence number of the receiver’s next octet to receive (RCV.NXT), and the size of the sender’s send window (SND.WND) is the same as the receiver’s receive window size (RCV.WND). RFC-1122 recommends using a sequence number (SEG.SEQ) of SND.NXT-1 with or without one octet of garbage data as the keep-alive segment. Using one octet of garbage data makes the keep-alive mechanism compatible with early TCP implementations that cannot handle a SEG.SEQ equal to SND.NXT-1 without one octet of data. Because the sequence number SND.NXT-1 lies outside the peer’s receive window, it causes the peer TCP to respond with an ACK segment if the connection is still alive; if the peer has dropped the connection, it will respond with a reset (RST) segment instead of an ACK segment [14].

3.2 Results

All the TCP implementations we tested correctly set the default so TCP did not send keep-alive probes, and let the applications turn on keep-alive in a per connection basis. Most implementations use a 7,200 second (2 hours) time interval between probes, as specified in RFC-1122. SunOS 4.0.3 uses a 75-second interval between probes. However, none of the implementations allow users to configure the probing interval as mandated in RFC-1122. Although Solaris 2.1 provides a *socket* option to turn on the TCP keep-alive, we did not observe any keep-alive probes in five observations; each observation lasted for 30 hours.

RFC-1122 does not specify the contents of the acknowledgment field (SEG.ACK) of the keep-alive segment. However, as Table 1 shows, most of the TCP implementations set the SEG.ACK to RCV.NXT-1. It is unnecessary to set SEG.ACK to RCV.NXT-1 unless it is also for backward compatibility with early TCP implementations. To see if probed implementations

respond to a keep-alive segment that has SEG.SEQ equal to SND.NXT-1, SEG.ACK equal to RCV.NXT, and does not include one octet of data, we modified the SunOS 4.0.3 TCP code to send such a keep-alive segment. All implementations responded correctly to the keep-alive segment.

3.3 Keep-alive and Server Applications

TCP keep-alive is especially useful for a server application to prevent clients from holding server resources indefinitely after clients crash or after a network failure. As an example to see how network failure can affect a host when a server application does not turn on TCP keep-alive and does not deploy mechanisms to handle idle connections, consider the probe procedure used in section 2. The probe procedure deliberately disables interface A on host H while a probe program on host T is communicating with the TCP discard server⁹ on host H via interface A. After host T retransmits a data segment for a preset number of times without any response, it closes the connection. Unfortunately, the discard server on host H has no idea that the peer has aborted the connection because it does not turn on the TCP keep-alive and makes no attempt to detect the idle connection. From its point of view the connection remains quiet. After each experiment, there is an *orphan* discard server process left on host H . These orphan server processes stay until the system reboots or a system manager destroys them explicitly¹⁰.

4 Zero-Window Probes

TCP in a receiving host uses the *window* field in each acknowledgement to inform TCP in the sending host how much more data it is willing to accept [14]. If the receiver temporarily runs out of buffer space, it sends an ACK with the window field set to zero. When space becomes available, the receiver sends another ACK with a nonzero window size. Because the ACK that reopens window can be lost in transit, the connection may hang forever. TCP specifications [1, 14] require a host that has received a zero window advertisement to transmit zero-window probe segments to the receiving host requesting its current buffer space if it does not receive a nonzero window advertisement in a specified period of time. The sender must increase the intervals between the zero-window probes exponentially as it does for retransmissions.

⁹The program `inetd` implements the discard server.

¹⁰To prevent too many orphan discard server processes from affecting the experiment, we destroyed the orphan process after each experiment.

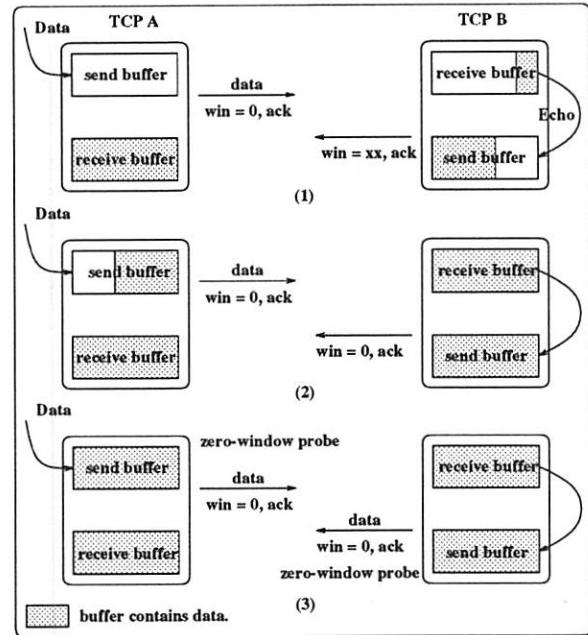


Figure 6: Generating zero-window probes using TCP echo service

4.1 Probing Procedure

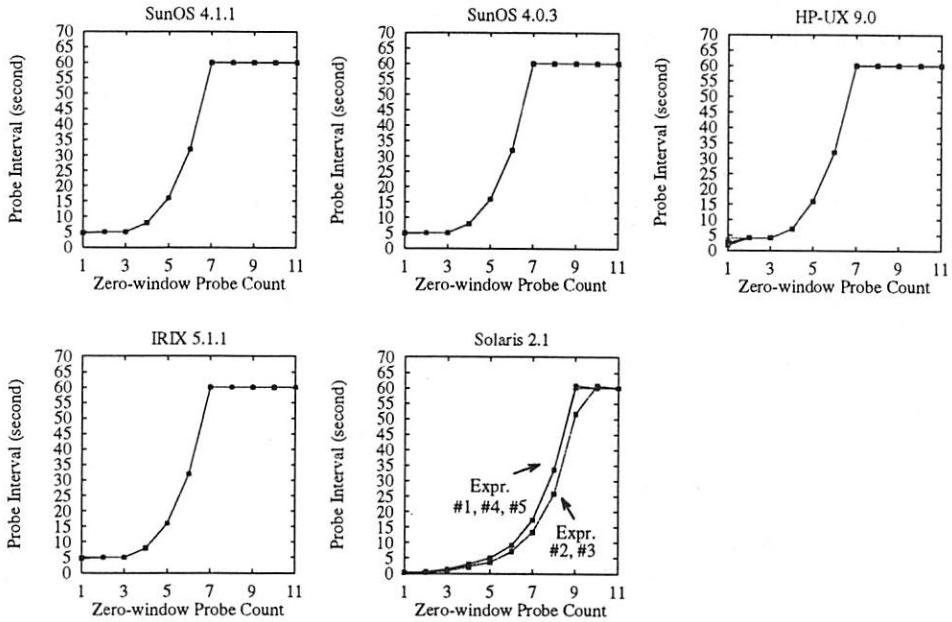
We use the following simple procedure to study zero-window probing in various TCP implementations. For each implementation, we conduct five experiments.

1. From a host to be tested, open a TCP connection to the *echo* port [15] of another host.
2. Keep sending data to the echo port without reading the echoed data.

As Figure 6 shows, because the probe program sends data without reading the echo, the receive buffer of TCP A eventually becomes full, causing it to send a zero-window ACK segment to TCP B. Because TCP B cannot send data to TCP A, the send buffer of TCP B will become full of echoed data. When the echo server on B cannot send more data, the receive buffer of TCP B will become full. Once the receive buffer of TCP B becomes full, it advertises a zero window to TCP A. After the zero-window condition exists for more than a threshold time period, both sides begin sending zero-window probes.

4.2 Results

As Table 2 and Figure 7 show, all the implementations probed exponentially increase the time interval



Note: Only the time intervals of the first 11 zero-window probes are shown.

Figure 7: The intervals of successive zero-window probes in TCP implementations

Operating System	Data size in 0-win probe seg.	Min. probe Interval	Max. probe Interval
Solaris 2.1	1 MSS octets	200 ms	60 sec.
SunOS 4.1.1	1 octet	5 sec.	60 sec.
SunOS 4.0.3	1 octet	5 sec.	60 sec.
HP-UX 9.0	1 octet	4 sec.	60 sec.
IRIX 5.1.1	1 octet	5 sec.	60 sec.

Table 2: Zero-window probe in TCP implementations

between probes and limit the probe interval to a maximum value of 60 seconds. Most implementations impose a minimum probe interval between 4 and 5 seconds; Solaris 2.1 uses the lower bound on RTO estimates as the minimum probe interval, which is much smaller than other systems.

Figure 7 shows another difference between Solaris implementation and other systems — there are two curves on the graph of Solaris. One curve corresponds to the results of two experiments (Experiment #2 and #3) and the other curve corresponds to three. A plausible explanation of the difference is that Solaris uses a finer granularity timer than other systems. If the probe intervals shown represent an exponential increase, divergence in the two curves must result from a difference in the initial RTO values. We conclude that Solaris 2.1 TCP had two RTO estimates during the experiments.

4.3 Two Approaches In Handling Zero-window Probing

From the data, we observe two approaches used to handle zero-window probing. Observe that a sender does not need to distinguish between a peer that has insufficient buffer space to receive a segment and a segment that is lost. In both situations, the data segment is unable to reach the application. Although a receiving TCP will generate a zero-window ACK segment when it has no receive buffer space and will not generate an ACK for a lost data segment, the unreliable delivery of the zero-window ACK segment in TCP makes both situations look similar to a sending TCP. The observation suggests that one can use a retransmitted data segment as a zero-window probe.

Indeed, the first approach uses a retransmitted data segment as a zero-window probe. If a receiving TCP does not have sufficient buffer space to accept an incoming data segment, it sends a zero-window ACK without acknowledging the data segment. After a period of one RTO, the sender retransmits the data segment. The retransmitted data segment acts as a zero-window probe. Unlike retransmitting missing data segments, a sender keeps transmitting zero-window probes even if a receiver does not ACK the probes.

Using a retransmitted data segment as a zero-

Segment Number	Host A (SunOS 4.1.1)	Host B (Solaris 2.1)	Comment
(Both side have zero receive window)			
1094		← S:8552 D:512 A:1369 W:0	(zero-window probe)
1095	S:1369 D:0 A:8552 W:0	→	(ACK with window = 0)
	(5 seconds later)		
1096	S:1369 D:1 A:8552 W:0	→	(zero-window probe)
1097		← S:9064 D:512 A:1369 W:0	(ERROR! bad sequence #)
1098	S:1369 D:0 A:8552 W:0	→	(ACK with the seq. # expected)
.....			

S: Sequence number, D: Number of data octets, A: Acknowledgment number, W: Window.
Note: Only the last four digits of the sequence number and acknowledgment number are shown.

Figure 8: Illustration of an implementation flaw in Solaris 2.1 TCP

window probe is optimistic in the sense that it sends as much data as possible in a zero-window probe and expects the receiver's receive window to open within one RTO period. The scheme responds quickly when an ACK that would reopen the window is lost. The scheme is also efficient because TCP implementations must implement Silly Window Syndrome avoidance algorithm¹¹[1, 2]. It is likely that when the receiver opens the receive window, it will open at least the size of a maximum segment (1 MSS). However, the scheme consumes more network resources than the second approach, described below, when the receiver's zero-window persists.

The second approach treats zero-window probing as a special case. When a sender receives a zero-window advertisement from the receiver, it enters a zero-window probing state and delays sending data for a predetermined interval τ ¹². If a window-opening ACK segment arrives within interval τ , TCP immediately sends data without sending zero-window probe. However, the scheme suffers a (long) delay of τ if an ACK segment to reopen the window is lost in transit. The zero-window probes in this approach carry only one octet of data; they are designed to elicit an ACK segment from the peer, not to transfer data.

From the experiments, we conclude that Solaris uses the first approach, and the others use the second approach.

4.4 Implementation Flaw Found

The data from zero-window probe experiments shows protocol violations in the SunOS 4.0.3 version and an implementation flaw in Solaris 2.1. SunOS 4.0.3 TCP does not acknowledge zero-window probes at all. Solaris 2.1 TCP responds incorrectly to a peer's zero-window probe when both sides have zero receive window; we describe the flaw below.

As Figure 8 illustrates, host A communicates with host B (running Solaris 2.1); both hosts have a zero receive window. In segment #1094, B sends a zero-window probe with sequence number 8552 and 512 octets of data to A. A acknowledges it properly in segment #1095. Five seconds later, in segment #1096, A sends a zero-window probe with one octet of data to B. Note that the ACK number in segment #1096 is the same as the ACK number in segment #1095, i.e., A did not acknowledge the 512 octets of data that B sent in segment #1094. However, B acknowledges the zero-window probe with a segment (segment #1097) containing an invalid sequence number 9064 (8552+512), as if the zero-window probe from A had acknowledged the segment it sent in segment #1094. A acknowledges the error by sending an ACK segment (segment #1098) with the sequence number it expects. The flaw occurs in all of the Solaris trace data we gathered.

5 Conclusion and Future Work

This paper introduces the active probing technique and demonstrates how it can be used to study TCP implementations. The technique treats a TCP implementation as a black box and uses specially designed probe procedures to examine its behavior. A packet trace taken during active probing can be used to de-

¹¹Silly Window Syndrome is characterized as a situation in which a steady pattern of small TCP window increments results in small data segments being sent. Sending small data segments lowers TCP performance because TCP and IP headers consume network bandwidth.

¹²Experiments show that τ is 4 or 5 seconds in the implementations probed (see Table 2).

duce design parameters and design decisions in TCP implementations. The results show that active probing is an effective tool.

Insight into black box behavior depends on probe procedure design and careful analysis of the resulting output. We demonstrated three probe procedures that examine three aspects of TCP. Additional probe procedures to study other aspects of TCP are also possible. For example, one can design a probe procedure that generates heavy network traffic through a gateway to examine how a TCP behaves in a congested environment.

Because active probing can be used to deduce design parameters and design decisions in TCP, the technique can also be applied to protocol conformance checking. One can design procedures that induce output from a TCP implementation, and use an automated tool to analyze the output and verify that it conforms to the protocol specification. For example, the failure to respond to the zero-window probes in SunOS 4.0.3, as discussed in section 4, can easily be detected by such a method.

The implementation flaws found also show that active probing can be used to test whether TCP implementations operate correctly. From the point of view of software engineering, one can design probe procedures to create conditions that occur frequently or infrequently, thus providing tests that cover cases not normally found through passive monitoring.

Unusual output can be used to detect implementation flaws in TCP. For example, an implementation of TCP that generates excessive retransmissions in a LAN environment may contain an implementation flaw. The implementation flaws in Solaris 2.1, as discussed in sections 2 and 4, were detected by observing excessive retransmissions in the trace output. It would be interesting to combine a knowledge-based trace analysis tool [5] with active probing to accurately detect other abnormal TCP behavior.

Finally, most of the TCP implementations probed in this paper are BSD derived TCP implementations. It is possible to probe non-BSD derived TCPs (e.g., Plan9 TCP [13, 17] and Xinu TCP [3]) to determine the similarities and differences.

6 Biographies

Dr. Douglas Comer is a full professor of Computer Science at Purdue University. He has written numerous research papers and textbooks, and heads currently several networking research projects. He designed and

implemented X25NET and Cypress networks, and the Xinu operating system. He is director of the Internet-working Research Group at Purdue, editor-in-chief of the *Journal of Internetworking*, editor of *Software – Practice and Experience*, and a former member of the Internet Architecture Board.

John Lin is a PhD student in the Department of Computer Sciences at Purdue University. He received the M.S. degree in Information and Computer Science from Georgia Institute of Technology in 1988. Before attending Purdue, he was a member of scientific staff of Bell-Northern Research. He received the Outstanding Teaching Assistant award from Purdue ACM in 1992 and a two-year fellowship from UniForum Association in 1993. His research interests include operating systems, transport protocol design and analysis, distributed systems, and high-speed networking.

7 Acknowledgment

The authors are grateful to Win Treese and the anonymous reviewers for their comments on an earlier draft of this paper.

8 Trademarks

UNIX is a registered trademark of UNIX System Laboratories, Incorporated. SunOS and Solaris are trademarks of Sun Microsystems, Incorporated. HP-UX and HP NetMatrix Applications are trademarks of Hewlett-Packard Company. IRIX is a trademark of Silicon Graphics, Incorporated. UniForum is a registered trademark of UniForum Association.

References

- [1] R. Braden. RFC-1122: Requirements for Internet Hosts – Communication Layers. *Request For Comments*, October 1989. Network Information Center.
- [2] David D. Clark. RFC-813: Window and Acknowledgement Strategy in TCP. *Request For Comments*, July 1982. Network Information Center.
- [3] D.E. Comer and D.L. Stevens. *Internetworking with TCP/IP Vol. II: Design, Implementation, and Internals*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
- [4] R. Enger and J. Reynolds. RFC-1470: FYI on a Network Management Tool Catalog: Tools for

- Monitoring and Debugging TCP/IP Internets and Interconnected Devices. *Request For Comments*, June 1993. Network Information Center.
- [5] Bruce L. Hitson. Knowledge-based monitoring and control: An approach to understanding the behavior of TCP/IP network protocols. In *Proceedings of SIGCOMM '88*, pages 210–221, Aug. 1988.
 - [6] HP - Metrix Network Systems. *NetMetrix V3.02 Protocol Analyzer and Load Monitor*, 1992.
 - [7] V. Jacobson, R. Braden, and D. Borman. RFC-1323: TCP Extensions for High Performance. *Request For Comments*, May 1992. Network Information Center.
 - [8] Van Jacobson. Congestion Avoidance and Control. In *Proceedings of SIGCOMM '88*, pages 314–328, Aug. 1988.
 - [9] Phil Karn and Craig Partridge. Improving Round-Trip Time Estimates in Reliable Transport Protocols. *ACM Transactions on Computer Systems*, 9(4):364–373, November 1991.
 - [10] S.J. Leffler, M.K. McKusick, M.J. Karels, and J.S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, Reading, Massachusetts, 1990.
 - [11] D. L. Mills. RFC-889: Internet Delay Experiments. *Request For Comments*, December 1983. Network Information Center.
 - [12] John Nagle. RFC-896: Congestion Control in IP/TCP Internetworks. *Request For Comments*, January 1984. Network Information Center.
 - [13] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell Labs. In *Proc. of the Summer 1990 UKUUG Conf.*, pages 1–9, July 1990.
 - [14] J. Postel. RFC-793: Transmission Control Protocol. *Request For Comments*, September 1981. Network Information Center.
 - [15] J. Postel. RFC-862: Echo Protocol. *Request For Comments*, May 1983. Network Information Center.
 - [16] J. Postel. RFC-863: Discard Protocol. *Request For Comments*, May 1983. Network Information Center.
 - [17] D. Presotto, R. Pike, K. Thompson, and H. Trickey. Plan 9, A Distributed System. In *Proc. of the Spring 1991 EurOpen Conf.*, pages 43–50, May 1991.
 - [18] D. P. Sidhu and T. P. Blumer. RFC-964: Some problems with the specification of the Military Standard Transmission Control Protocol. *Request For Comments*, November 1985. Network Information Center.
 - [19] Lixia Zhang. Why TCP timers don't work well. In *Proceedings of SIGCOMM '86*, pages 397–405, Aug. 1986.

Experiences with a Survey Tool for Discovering Network Time Protocol Servers

James D. Guyton, Michael F. Schwartz
University of Colorado, Boulder

Abstract

The Network Time Protocol (NTP) is widely used to synchronize computer clocks throughout the Internet. Existing NTP clients and servers form a very large distributed system, yet the tools available to observe and manage this system are fairly primitive. This paper describes our experiences with a prototype tool that attempts to discover relevant information about every NTP site on the Internet. The data produced by this tool can be used for a variety of purposes, including locating nearby accurate time servers and computing aggregate and long-term evaluations of the size and health of the NTP system. We are building a client/server system around this tool, to allow new NTP server administrators to make informed choices among the possible servers with which to synchronize, balancing the need for accurate time with the need to distribute NTP server load. This is an important step towards improving global NTP system scalability, since at present our measurements indicate that the high-stratum servers are heavily overloaded.

1. Introduction

1.1. Clock Synchronization and NTP

Clock synchronization is useful for a wide variety of purposes, particularly in a network environment. Uses include keeping accurate file timestamps in distributed filesystems (e.g. so that *make* doesn't get confused), ticket validation timers in security systems like Kerberos [SNS88], and potentially even synchronizing clocks across the globe for very-long baseline radio astronomy work.

Designed and developed by David Mills of the University of Delaware, the Network Time Protocol (NTP) [MIL92] is currently used by thousands of Internet hosts to synchronize their local clocks to within a few milliseconds of the international time

standard. Systems to distribute accurate time have been studied for some time [LIN80, BRA80], but NTP's contribution is that it is able to distribute accurate time over the unpredictable Internet. Hosts participating in time distribution via NTP form a hierarchical, master-slave, self-organizing subnet¹ of the Internet. At the top layer of the hierarchy are the sources of very accurate time. These *stratum-1* servers usually have local atomic clocks (that are synchronized by means other than NTP) or radio receivers that decode time signals broadcast by national standards organizations (e.g. WWV timecodes broadcast by NIST in Ft. Collins, Colorado).

NTP goes to great lengths to distinguish time servers with accurate data from those with false data and to obtain the best synchronization possible in the face of widely varying Internet delays. Recent versions of NTP include algorithms to combine the offsets of several clocks, to construct a synthetic time more accurate than any individual time server.

In addition to its value for supporting clock synchronization, running an NTP server can indirectly help network managers uncover and diagnose network problems, based on the statistics NTP maintains. For example, NTP keeps a status register of how "reachable" its servers have been recently. This information provides a crude but useful measure of packet loss, which can be helpful in diagnosing network load or connectivity instabilities.

We chose to study NTP because it is an important and widely distributed system. At present, it is "bundled" with the software distributions from a number of workstation manufacturers, and a number of large organizations use NTP. The U.S. Weather

¹The word "subnet" is used in this paper to refer to the subset of Internet hosts that use the NTP protocol. It should not be confused with the more common usage, which denotes partitioning a single IP network number into multiple smaller networks.

Service uses NTP, and soon every Public Broadcasting Service station affiliate will be a client. Merrill Lynch is currently populating its worldwide network with NTP.

1.2. NTP Management Problems

While NTP goes to great lengths to maintain well-synchronized clocks in the face of unpredictable Internet behavior, at present managing the NTP network itself is quite difficult. The latest NTP software distribution includes a few debugging tools for examining and changing the state of an individual server, but does not include tools to discover the nearest NTP server or to help debug the NTP system as a whole. When only a few sites ran NTP, this information was easy to gather by manual methods. But now that there are many thousands of sites running some version of NTP, the need for additional discovery and management tools has become painfully clear.

The largest problem is that the stratum-1 servers are seriously overworked and in danger of becoming saturated. In principle this should not be a problem, because NTP allows time to be distributed hierarchically. If this hierarchical architecture were used appropriately, then the NTP protocol is theoretically capable of distributing accurate time to every host on the Internet.²

The problem arises because it is not trivial for an NTP server administrator to pick an appropriate set of servers with which to "chime" (i.e., synchronize clocks) when first joining the NTP server network. As a result, far too many administrators select high-level NTP servers.³ An NTP discovery system that allowed new administrators to identify "nearby" servers would reduce this problem, and markedly enhance global system scalability. Running periodic surveys would also make it possible for regional network administrators to monitor the configuration of the NTP subnet within their domain and apply social pressure to fix poorly configured NTP hosts.

An alternative approach would be to restrict which systems could chime with high-stratum NTP servers, through an access control mechanism. While the code for this approach has been implemented, it has not been widely adopted. In our opinion this approach should be avoided – it would result in more work for the administrators of the restrictive servers, while merely increasing the load on the remaining

²For example, assuming only 10 stratum-1 servers and a very light load of 10 clients per server, the NTP subnet could contain a maximum of 10^{15} hosts. This is much larger than the current IP address space.

³See the following section on Survey Results, and Figure 2 for details.

unrestricted servers. Because the problem arises from the inability to obtain good information, a solution based on discovery seems more appropriate than one based on access control.

In addition to supporting more well-informed configuration management choices, an NTP discovery/survey tool is useful for helping to debug and understand the NTP protocol itself. To our knowledge there has never been an aggregate picture of the "state of the NTP network" at a finer level of detail than a simple count of the number of hosts running NTP. We hope that by offering the ability to collect meaningful measurements of the state of the NTP world, a deeper understanding of the workings of this large, distributed system can be attained, which would enable further improvements to the NTP model.

2. Survey Methodology

There is no complete registry of hosts that run NTP. Instead, we begin with a list of hosts known to run NTP, query each host with an NTP information request, parse the response, and add any previously unseen hosts to the database. These newly discovered hosts are then queried and the process repeats until no new hosts are discovered.

At first examination, one might believe that this iterative survey process would quickly discover all Internet NTP hosts. For a variety of reasons, this is not so. In the following sections we discuss the problems and the prototype's approach to solving them.

2.1. Monitor List Queries

The biggest difficulty in implementing an NTP survey is that the protocol does not require each NTP time server to keep track of its clients. While all NTP implementations require clients to track the server with which they chime, they may keep little or no information about clients that chime with them. This makes it easy to move up the NTP hierarchy, but often impossible to move down. Given that we know the most about the top two levels of the tree and relatively little about the leaves, this is a serious restriction.

While keeping track of clients is not required by the NTP specification, it is frequently very useful for debugging. Client tracking is supported by the XNTP implementation of NTP Version 3 [XNTP93]. This debugging feature of XNTP is called "monitor mode", and the results of monitoring can be retrieved remotely with the "monitor list" command.

2.2. System and Peer Variable List Queries

Versions 2 and 3 of the NTP protocol specify a standard control packet format that allows internal NTP variables to be examined and set. These variables contain useful information about the state of the local NTP system, the software phase-locked-loops and filters that it uses, the status of the peers with which a host is chiming, and a wealth of other information. These queries provide a good source of information for an NTP survey.

An immediate problem with an NTP survey tool is to choose what variables should be requested. The current solution to this problem is to allow the user to specify the set of interesting variables. A configuration file lists all of the system and peer variables to be used in queries, and any returned values for these variables are recorded by the survey.

One problem with variable queries is that different implementations of NTP can have slightly different spellings of some of the variable names, resulting in query failures when a server is asked for non-existent variables. What makes this minor problem much worse is the fact that, while a variable query command can contain a long list of variable names whose values are to be returned, all observed NTP implementations simply return an error if *any* of the variable names are unknown to that implementation. This leads to the need for a rather complex query/error/retry algorithm to extract data from each NTP server.

A smaller problem with variable queries was caused by the fact that query responses are returned formatted for human-consumption, including white space, punctuation and newlines. While this response makes it easy to write simple query/display programs, it made implementation somewhat more difficult for our NTP survey tool.

2.3. Version Issues

The NTP protocol is an evolving entity, and various implementors have made substantial improvements with four major versions of the protocol over the past eight years [MIL85, MIL88, MIL89, MIL92]. But as with any widely deployed system, there have been a few compromises made to facilitate backwards compatibility.

The basic idea is that newer versions of NTP may interoperate with systems running older versions, but when they do so it is suggested that they "fib" about their own version number so as not to confuse the remote system. This is wonderful from a compatibility point-of-view, but makes it difficult for our survey tool to determine correct version numbers.

2.4. Access Restrictions

A potential problem that we ignored in our current prototype is the issue of authentication. Given that a malicious user could try and skew clocks by supplying faulty NTP times, the NTP protocol specification includes support to authenticate requests. Unfortunately, an NTP server that uses authentication is not queryable by our discovery tool. Even though the authentication code is widely deployed, the need for it has not become severe, and at least for the moment an NTP explorer can blissfully explore the network without worrying about lack of permissions and magic keys.

One reason NTP-based access controls are not often used is that sites often use firewall gateways [CQ92] to control all incoming traffic, rather than setting up restrictions on a per-service basis. These gateways present difficulties for our survey tool because there is no easy method of determining whether a potential NTP host is behind a firewall. From the survey tool's perspective the host appears to be "temporarily" unreachable. The only solution we can see for this problem is to add survey functions to the firewalls so that (approved) statistics of what's going on behind the firewall can be exported to survey tools like this one.

One final access restriction note concerns a lesson learned by other network information discovery projects: Some Internet system administrators consider network discovery methods to be distressingly equivalent to trespassing. Fortunately, this is such a small percentage of the Internet community that their hosts can safely be omitted from the survey without seriously affecting the survey results. The prototype NTP explorer module has a simple method of host and network avoidance. We initialized its "don't trespass" database from a list of systems whose administrators had previously requested to be left alone.

3. Implementation

The current implementation consists of two query programs written in C, and a small collection of utilities and filters written in C and PERL. The two query modules both take a list of IP addresses as input, do either a monitor-list or variable query, and update the databases with their results.

There are two very different types of data that need to be collected by the NTP survey tool. One type of data is the NTP topology and the set of hosts that participate in the NTP subnet. These data are relatively small and easy to manage.

The other type of data that need to be collected by an NTP survey is a large amount of NTP state

information giving the status of the NTP protocol engine at each reachable NTP host in the Internet. These data need to be recorded and made available to analysis tools to understand how the NTP network is performing and changing over time.

The current implementation splits these two types of data into different databases. One is a standard UNIX "dbm" file that contains minimal information about each host that is an actual or suspected participant in the NTP subnet, along with the timestamps and status codes of recent NTP query attempts. The other is a relational database recording all the detailed NTP data that are gathered by a particular survey run. These later data are stored in an RDB [HOB93] database, which supports a simple relational model that eases manipulation and analysis.

We seed the survey database from a list of publically available stratum-1 and stratum-2 time servers manually maintained by Mills.⁴ At present this file lists approximately 35 stratum-1 servers and 70 stratum-2 servers. Since this file is not designed to be parsed electronically, we manually extracted time server host names from this file, and used them to seed the survey database. This initialization program is the only module in the prototype that allows the use of full domain-style hostnames. The rest of the implementation simply uses IP addresses in the interest of performance. Additional hosts to check can be added to the database by hand or by other means (e.g. a Fremont explorer module that has reason to believe that a host may be chiming NTP; see the Related Work section for a discussion of Fremont).

4. Survey Experiences and Results

The monitor list query uses a packet format that the NTP Version 3 standard defines only as "reserved for private use." Fortunately, the monitor list command is often available, because it is included in the widely deployed XNTP implementation. However, not all sites that run NTP use XNTP, and many that do run XNTP leave monitor mode disabled (it is disabled by default). Moreover, as mentioned earlier, some of those that collect monitor data require prior authorization to use the "monitor list" query and retrieve the information.

Even with this daunting list of restrictions, it turns out that there are enough publically retrievable monitor listings that using this style of query resulted in gathering evidence of approximately 10,000 possible NTP hosts in about 8 hours of survey time.

⁴Available by anonymous FTP from louie.udel.edu, /pub/ntp/doc/clock.txt

By comparison, the survey module that queries for system and peer variables plods along gathering a great deal of information, but adds comparatively few hosts to the database of potential NTP systems. It took about 50 hours to attempt to query 10,000 hosts.⁵

The number of NTP hosts found in the initial survey was relatively large. The main database contains over 15,000 unique IP addresses of hosts that we have reason to believe speak NTP, and the survey was able to speak NTP directly with over 7,200 systems. While this database contains no duplicate IP addresses, hosts with multiple network interfaces may be counted twice.

In comparison with other NTP surveys, our survey has done rather well. Mills' survey of July 1993 [MIL93a] found a total of 6,185 hosts (via monitor list), while a survey done by Pruy in October 1993 [PRUY93] was able to communicate with about 2,100 hosts.

The results of an NTP survey are a little tricky to summarize without being misleading. The set of NTP hosts in the database built by recursively running the "monitor list" command is much larger than the set of NTP hosts that are actually reachable by NTP information queries. Therefore, we distinguish the results below by data source.

4.1. Statistics from "Monitor List" Queries

The monitor facility of XNTP records information about a particular server's clients. The IP addresses and NTP protocol version number of its clients are recorded, but very little else is recorded. As described before, the version number can be fictitious, and must be taken with a dose of skepticism.

The total number of IP hosts derived from 1,760 hosts that responded to monitor lists queries are listed in Table 1. It is interesting that, although the majority of sites are running the most recent version

Version 3	7,615
Version 2	2,432
Version 1	2,095
Version 0	58
Total Hosts	12,200

Table 1: NTP Host Count

⁵The long survey times are a result of the current implementation's use of sequential reads and timeouts.

of NTP, many sites still have not upgraded. Part of the problem is that a number of workstation manufacturers are bundling outdated versions of NTP [MIL93b]. It would be interesting to collect these measurements periodically as the NTP subnet continues to grow, and see what percentage of "old" hosts upgrade vs. how many new hosts start by running the latest version.

4.2. Statistics from "Variable List" Queries

The data from the peer and system variable queries are more accurate than those from monitor list queries, though not nearly as complete. Even so, an amazing amount of raw data from our survey is available for analysis. The following statistics are but a first-pass at mining interesting information from it.

While there were over 15,000 hosts in the completed survey's database of suspected NTP hosts, only 7,251 responded to NTP variable list queries. The statistics in Figure 1 are summarized from the data returned by these 7,251 hosts. Clearly, the high-level strata are overused. At present, Mills attempts to reduce this problem periodically by sending a message on the NTP mailing list asking people to back off of the stratum-1 servers and to make more use of the stratum-2 servers. Perhaps if our NTP discovery tool were built into the system (so that new site administrators could choose a good peer with which to chime), this would be less of a problem.

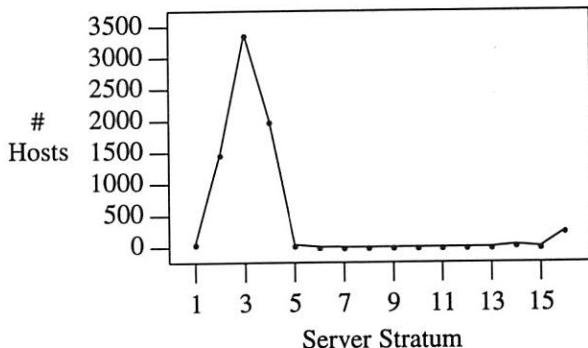


Figure 1: NTP Hosts per Stratum

Note that stratum 16 is defined by the NTP protocol specification as infinitely far away from the time source. The hosts that claim to be at strata 13 through 15 have more subtle problems. At first examination they appear to be a collection of isolated hosts in Germany with their own time source that believes itself to be at stratum-13, plus an extremely confused set of hosts at the University of Tennessee.

The average number of clients per server can be directly computed from the above figure, and are shown in Figure 2. This figure clearly indicates how poorly the stratum-2 (and lower) servers are utilized.

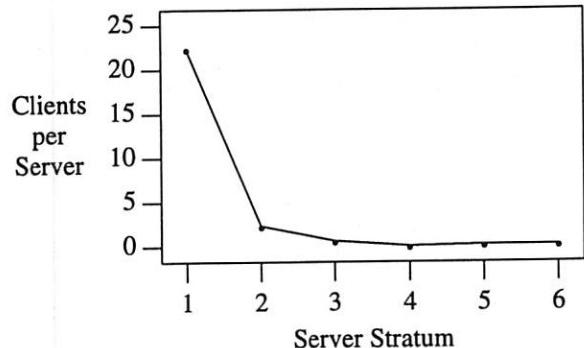


Figure 2: NTP Tree Branching

Central to the NTP clock calculations are the delay and dispersion values between a client and its server. *Delay* is the round-trip network delay between the peer and its server. *Dispersion* is the computed maximum error of the peer clock relative to its server. Both quantities are typically measured in milliseconds. Table 2 shows some statistics regarding the aggregate delay and dispersion between clients and their servers, grouped by the stratum level of the servers.

Rootdelay is the estimated total delay to the top of the NTP tree, while *rootdispersion* provides an error bound for how far off the local clock is from the NTP stratum-1 time source. Rootdispersion is probably as good a metric to estimate the health of the NTP subnet as any. Table 3 summarizes the rootdelays and rootdispersions from clients at each stratum to their NTP time source.

Tables 2 and 3 show that there is a fairly small median error as the time is distributed from stratum to stratum, with the average being substantially higher. In other words, many low-stratum servers offer very good time, but some offer very bad time.

Without advance knowledge of "how good" a set of time servers might be, people tend to pick high-stratum servers, because servers at the top of the distribution tree would intuitively seem to provide more accurate times. Yet, as Tables 2 and 3 show, this is not necessarily the case. In fact, because of the new NTP clock-synthesis code, it is actually possible that a low-stratum clock may be more accurate than any of its parents in the distribution tree. Tools are needed so that new NTP server administrators can make informed choices among the possible servers with which to chime, so that they can pick a

server that provides accurate time without overloading the high-level servers.

5. Observations

After examining the results of our survey, a number of unexpected results emerged. The most surprising are enumerated below:

- A large number of hosts run NTP version 1. Out of the 15,000 hosts in the database, the NTP survey tools were able to speak NTP with about 7,200 hosts. About 7,700 of the remaining hosts did not respond to either Version 2 or Version 3 NTP packets. A small test program revealed that 5,600 of these hosts were reachable⁶, and of these 2,350 hosts would respond to an NTP version 1 date query.
- Many hosts appear in monitor lists that do not run NTP servers. The numbers mentioned above imply about 3,300 hosts that are reachable and are suspected of running NTP (i.e., appear in the database for one reason or another), do not run a full NTP peer. The most likely conclusion is that these hosts use NTP to set their clocks when booting, but do not run the normal NTP server process.

- After sending a query to an NTP host, the response would often come back from an IP address different than what was expected. The NTP code currently treats this as an error, but it should instead be treated as a serendipitous discovery of multiple network interfaces on a gateway.
- While the initial implementation had a simple filter to delete the "loopback" host and other obviously invalid IP addresses before adding them to the database, about 85 of the 15,000 "hosts" still turned out to be network numbers. It would be useful to track down how these network numbers were introduced.

5.1. The Need for Stability Measures

One final observation is that the current NTP tools provide a way to observe the time errors *at the time of the query*. However, because NTP periodically resynchronizes clocks, these measures may not capture instabilities. For example, there was a problem with the latest experimental version of NTP when run on HP workstations, that occasionally caused the clock to be incorrect by about 40 years. If a downstream client synchronized with such a clock after this error arose, it could lead to many problems. A

Clients to Stratum	Delay Average	Delay Median	Delay Std. Dev.	Dispersion Average	Dispersion Median	Dispersion Std. Dev.
1	105	79	111	38	20	87
2	42	30	74	46	18	164
3	36	39	62	55	21	125
4	42	43	19	114	73	153
5	50	53	19	112	91	83

Table 2: Client-to-Server Metrics [msec.]

Clients at Stratum	Root Delay Average	Root Delay Median	Root Delay Std. Dev.	Root Dispersion Average	Root Dispersion Median	Root Dispersion Std. Dev.
2	155	95	177	102	52	213
3	160	104	166	175	114	251
4	184	105	163	195	145	259
5	116	28	182	362	166	419

Table 3: Client-to-Root Metrics [msec.]

⁶The program simply tested whether it could get a response from the host's UDP echo port.

global NTP survey/management system should provide measures of clock error that would allow potential clients to avoid such instabilities.

5.2. What was Missed?

What percentage of the total do these numbers reflect? It's impossible to know for sure, since there are many firewalls on the Internet hiding an unknown number of NTP hosts. Dave Mills has estimated that there are approximately 100,000 hosts running NTP, tens of thousands of which are behind firewalls [MIL93b]. The only way to know for sure is to get the cooperation of the firewall sites to allow surveys into their domains. In the mean time, modifying XNTP to enable the monitor-list feature by default would quickly extend the reach of this survey tool.

6. Related Work

There are several systems related to various aspects of the current work. Census is a tool that recursively descends the Domain Naming System (DNS) tree, gathering information from as many sites as possible [GAN92]. While both Census and our survey tool attempt to discover a distributed collection of servers, the task is more difficult for NTP servers because not all servers track the clients they support. DOC is a tool that tests remote DNS servers for various configuration errors [HOT90]. While our survey tool focuses on performance problems caused by distribution tree imbalance, DOC uncovers incorrect DNS server configuration information.

Archie gathers directory listing information from "anonymous FTP" servers around the Internet, for use as an indexing/search service [ED92]. Archie is intended primarily as a location service, not for detecting problems.

The Simple Network Management Protocol (SNMP) defines a general method to query and control network servers [SNMP90]. If the NTP system were being written today, it would probably use SNMP instead of its native query/control protocol. Perhaps in time NTP will be changed to use SNMP, at which point the NTP survey code can begin to use more standard network management tools.

The Fremont system gathers and cross-correlates data from a number of network protocols and information sources to construct a picture of key network characteristics, such as hosts, gateways, and topology [WCS93]. Each source is collected by a distinct "explorer module", which deposits the gathered data in a network accessible database managed by a *journal server*. The invocation of explorer modules is under the control of a *discovery manager*, which decides what information needs to be collected

and which explorer modules should be invoked to collect the data.

Fremont's support for multiple explorer modules, a discovery manager, and journal server, makes the opportunity for synergistic results obvious. The data mined by the current NTP survey tool should eventually be turned into data for the Fremont system.

7. Conclusions

In this paper we presented a survey tool for discovering the topology of the global NTP server network, and a set of experimental results from running this tool on the Internet. The basic survey methodology involves iteratively expanding a seed list of NTP hosts, based on information retrieved by NTP queries. This approach is complicated by a number of operational issues in the NTP network, ranging from version mismatches to firewall gateways and other limitations.

Our survey provides measurements of the current size and configuration of the NTP server network, uncovering approximately 10% of the total number of estimated hosts running NTP. The survey was limited primarily by the presence of firewall gateways in the Internet. The results showed that the primary time servers are overloaded, and that the global NTP time distribution tree is very poorly balanced.

To allow continued growth of the NTP community, a more balanced time distribution tree is necessary. Our discovery tool is a first step in developing the tools needed to help extend the scalability of the NTP system. In particular, Tables 2 and 3 demonstrate that the times from low-stratum servers often provide accurate times. Our tool provides a means by which new NTP server administrators can make informed choices among the possible servers with which to chime.

While the total extent of the NTP subnet probably won't be known until more hosts enable the "monitor list" command, the usefulness of this tool may encourage more people to do so. In the meantime, the current sample size seems large enough to collect more than enough information to make it interesting – and hopefully useful – to a wide variety of users.

One might observe that a discovery tool would not be necessary if NTP itself kept records of the server network topology (e.g., maintaining both upwards and downwards pointers, as is done in the DNS tree). There are at least two general reasons NTP does not provide adequate support in this regard. First, when developing any complex

algorithm, it behooves software designers to focus on the problem at hand. In retrospect it is usually easy to point out what "should" have been planned into the system from the start, yet it makes sense to postpone worrying about issues that will only arise once a system becomes "wildly successful". Second, it often is not clear from the start what state information should be collected to aid in system management.

From this perspective, our discovery and survey tool provides both a "transition path" towards a future management-oriented version of NTP, and a set of experiences concerning what is needed.

8. Future Work

8.1. Client/Server Architecture for Exporting Data

At present we are developing an architecture within which to deploy our survey tool. In this system, a set of servers will explore and measure the NTP network from various parts of the Internet, and allow survey clients to pose queries concerning which NTP server they should select for clock synchronization. The architecture will need to provide topology-dependent answers, so that NTP clients are paired with nearby NTP servers. We will also provide an interface that accepts a set of network numbers (e.g., all the networks in a regional network), and displays NTP configuration data about the hosts within that region. With this tool one could quickly check if too many hosts are chiming outside their local region.

We will make the above software available by anonymous FTP from [ftp.cs.colorado.edu](ftp://ftp.cs.colorado.edu) in /pub/cs/distrib/chimesurvey, around the end of Summer 1994.

8.2. Performance Improvements

Even though the number of packets sent to do the NTP queries is fairly small,⁷ our tool currently takes a long time to complete a survey. The tool could be sped up significantly by using a simple "pipelined" design that allowed for overlapping I/O with timeouts. The major complications are parallel updates to the databases, and the fact that the NTP protocol is based on UDP (and hence the operating system interface is full of dangers of dropping packets). Neither of these are serious obstacles, and a fully parallel version of this explorer should be implemented.

⁷About 5-10 packets per host.

8.3. Additional Sources of Potential NTP Hosts

Our survey's success depends critically on what hosts were initially seeded in the survey database. To improve the survey's thoroughness, there should be other methods of seeding this database. Some possible methods include Ethernet monitoring logs; queries of selected hosts found in the DNS⁸; hostnames from messages posted to the netnews group that discusses NTP; hosts discovered from DNS traversals that contain "ntp" in their name; and maybe even checking hosts that have retrieved the NTP source using FTP from [louie.udel.edu](ftp://louie.udel.edu).

8.4. Integration with Fremont

Our NTP survey tool should be integrated with the Fremont discovery system [WCS93]. The NTP configuration program(s) could make use of the topology of the network that is recorded in the current Fremont database, and the NTP survey occasionally discovers hosts with multiple network interfaces that would be of interest to Fremont.

9. Acknowledgements

We would like to thank Dave Mills for helpful suggestions and comments about this work.

This work was supported in part by the National Science Foundation under grant numbers NCR-9105372 and NCR-9204853, the Advanced Research Projects Agency under contract number DABT63-93-C-0052, and an equipment grant from Sun Microsystems' Collaborative Research Program.

The information contained in this paper does not necessarily reflect the position or the policy of the U.S. Government or other sponsors of this research. No official endorsement should be inferred.

10. References

[BRA80] Braun, W.B., Short term frequency effects in networks of coupled oscillators. IEEE Trans. Communications COM-28,8 (August 1980), pp. 1269-1275.

[CQ92] S. Carl-Mitchell and J. S. Quarterman. Internet Firewalls. UNIX/World, 9(2), pp. 93-102, Tech Valley Publishing, Mountain View, California, February 1992.

⁸A simple heuristic might be to look for an NTP server running on routers and mail servers, and ignore systems that appeared to be Macs or PCs.

[ED92] A. Emtage and P. Deutsch. Archie - An Electronic Directory Service for the Internet. Proceedings of the USENIX Winter Conference, pp. 93-110, San Francisco, California, January 1992.

[GAN92] Ganatra, N., CENSUS: Collecting Host Information on a Wide Area Network, University of Santa Cruz, technical report UCSC-CRL-92-34, Anonymous FTP from ftp.cse.ucsc.edu.

[HOB93] Hobbs, W.T., RDB Software Distribution. Available by anonymous FTP from rand.org, pub/RDB-hobbs/RDB-2.5j.tar.Z.

[LIN80] Lindsay, W.C., and A.V. Kantak, Network synchronization of random signals. IEEE Trans. Communications COM-28,8 (August 1980), pp. 1260-1266.

[MIL85] Mills, D.L., Network Time Protocol NTP. ARPA Network Working Group Report RFC-958, University of Delaware, September 1985.

[MIL88] Mills, D.L., Network Time Protocol version 1 specification and implementation. ARPA Network Working Group Report RFC-1095, University of Delaware, July 1988.

[MIL89] Mills, D.L., Network Time Protocol version 2 specification and implementation. ARPA Network Working Group Report RFC-1119, University of Delaware, September 1989.

[MIL92] Mills, D.L., Network Time Protocol (Version 3) Specification, Implementation and Analysis. ARPA Network Working Group Report RFC-1305, University of Delaware, March 1992.

[MIL93a] Mills, D.L., Electronic mail of July 21, 1993 to the NTP electronic mailing list.

[MIL93b] Mills, D.L., Private communication on December 22, 1993.

[HOT90] Hotz, S., and Mockapetris, P, Automated Domain Testing. USC Information Sciences Institute, August 1990. Preliminary notes.

[PRUY93] Pruy, R., Electronic mail of October 28, 1993 to the NTP electronic mailing list.

[SNMP90] Schoffstall, M., Fedor, M., Davin, J., Case, J., A Simple Network Management Protocol (SNMP). ARPA Network Working Group Report RFC-1157, May 1990.

[SNS88] J. G. Steiner, B. C. Neuman and J. I. Schiller. Kerberos: An Authentication Service for

Open Network Systems. Proceedings of the USENIX Winter Conference, pp. 191-201, February 1988.

[WCS93] Wood, D.C.M., Coleman S., and M. Schwartz, Fremont: A System for Discovering Network Characteristics and Problems. Proc. Winter USENIX Conference, pp. 78-89, January 1993.

[XNTP93] Mills, D.L., XNTP 3.3 Software Distribution. Available by anonymous FTP from louie.udel.edu, pub/ntp/xntp3.3.tar.Z.

Author Information

James D. Guyton received his BA in Computer Science from the University of California at Santa Barbara in 1975. Since then he has worked at Xerox and the RAND Corporation. He has recently returned to graduate school and is working on an MS in Computer Science. Guyton can be reached by US Mail at the Computer Science Department, University of Colorado, Boulder, CO 80309-0430; or by electronic mail at guyton@cs.colorado.edu.

Michael F. Schwartz received his PhD in Computer Science from the University of Washington in 1987. He is currently an Associate Professor of Computer Science at the University of Colorado, Boulder. His research focuses on issues raised by international networks and distributed systems, with particular focus on resource discovery and network measurement. Schwartz chairs an Internet Research Task Force Research Group on Resource Discovery and Directory Service, and is on the editorial boards for *IEEE/ACM Transactions on Networking* and for *Internet Society News*. He is also a Guest Editor for an upcoming special issue of the *IEEE Journal of Selected Areas in Communication*, focusing on the Global Internet. Schwartz can be reached by US Mail at the Computer Science Department, University of Colorado, Boulder, CO 80309-0430; or by electronic mail at schwartz@cs.colorado.edu.

Profiling and Tracing Dynamic Library Usage Via Interposition

Timothy W. Curry

Internet: tim.curry@sun.com

Sun Microsystems, Inc.

2550 Garcia Ave.

Mountain View, CA 94043

Abstract

Run-time resolution of library functions provides a rich and powerful opportunity to collect workload profiles and function/parameter trace information without source, special compilation, or special linking. This can be accomplished by having the linker resolve library functions to special wrapper functions that collect statistics before and after calling the real library function, leaving both the application and real library unaltered. The set of dynamic libraries is quite large including interesting libraries like libc (the C library and Operating System interface), graphics, database, network interface, and many more. Coupling this with the ability to simultaneously trace multiple processes on multiple processors covering both client and server processes yields tremendous feedback. We have found the amount of detailed information that can be gathered has been useful in many stages of the project life-cycle including the design, development, tuning, and sustaining of hardware, libraries, and applications.

This paper first contrasts our extended view of interposition to other profiling, tracing, and interposing techniques. This is followed by a description and sample output of tools developed around this view; a discussion of obstacles encountered developing the tools; and finally, a discussion of anticipated and unanticipated ways those tools have been applied.

1. Motivation

The tools described in this paper were created to analyze performance of graphics applications. The application writers seldom has access to the graphics library source or profiled versions of the graphics libraries. The library and hardware provider seldom has access to application source and data files. Our goal was to get useful performance data without special requests placed on either the application or libraries.

We were also after more information than is typically available. While traditional profile tools might tell you how many times a line drawing function is called, they don't tell you what percentage of the lines are write-only versus read-modify-write operations; what the average length, width, and angle of the lines are; and what line styles are used. One can envision similar questions for a database, such as percentages of read versus write transactions, if access patterns were sequential or random, etc. This additional information significantly improves the ability to perform more detailed analysis and make more informed decisions.

Graphics libraries remain our group's primary interest but the tools are generic to any dynamic library and have been applied both internally and externally to profile, trace, and generally interpose on non-graphics libraries. Additionally, the data that can be collected has proven useful well beyond performance analysis.

2. Terminology

For detailed discussions on dynamic linking, the reader should refer to [1,2,3]. The techniques described in this paper presume the applications which are to be profiled and/or traced have already made the decision to use dynamically linked libraries and that the run-time linker/loader provides a means to perform interposition. System V release 4 (SVr4) UNIX and all versions of Solaris provide that means through an identical interface. This technology has been around for several years, is easily used, and commonly found in operating systems. A brief description of the terms used by this paper should quickly resolve various operating system terminology issues.

A *dynamic library* consists of a set of variables and functions which are compiled and linked together with the assumption that they will be shared by multi-

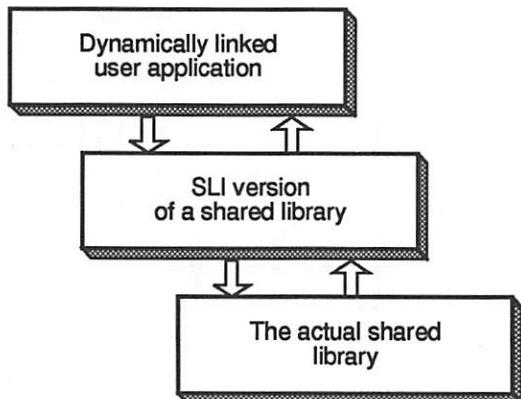
ple processes simultaneously and not redundantly copied into each application. Because of this sharing, dynamic libraries are sometimes called *shared libraries* or *shared objects*. Compiler and linker flags assure the program sections and data sections are cleanly separated and the program sections are reentrant and reusable. The compile and link of the application can leave the dynamic library *symbols* (variable and function addresses) unresolved until *run-time* (or time of execution). This complicates the *loader* (the program which initiates an application) since the *linking* (resolution of all symbols) must be completed on execution rather than at compile/link time.

The process of placing a new or different library function between the application and its reference to a library function is called *interposing*. We specifically avoid placing any constraint on what the interposing function must do other than accept the parameters of the real function and return an appropriate value back to the application. The real function may or may not be called and new side effects may or may not occur due to the interposing function.

3. Profiling and Tracing Techniques

We have developed a set of tools under the umbrella name SLI (pronounced sly) which is an acronym for Shared Library Interposer. SLI contains programs and utilities that enable application and library developers to monitor and analyze calls to shared library functions. SLI is intended to augment, not replace, analysis tools such as *tcov* [4,5,6,7], *gprof* [4,5,7], and *analyzer* [10]. This section provides an overview of our technique and contrasts it to other techniques.

3.1. Overview of SLI's Technique



What distinguishes SLI from traditional analysis tools is *how* it collects information and *what* infor-

mation it collects. The loader waits until execution of an application to resolve the shared library functions called by the application. SLI has the loader resolve the addresses to a wrapper of the library function, collects information in the wrapper, and calls the real library function from the wrapper. Several advantages arise from this technique:

- An accurate trace of call sequences can be logged.
- Parameter values are available to be logged and/or altered both before and after the real function call.
- The real function can be replaced if desired.
- Any subset of the library functions can be profiled instead of all functions in a library.
- Nesting levels into the libraries can be controlled.
- Different levels of profiling can be enabled or disabled while the application is running.
- Multiple processes and multiple library statistics can be logged to single or multiple locations.
- Profiling is available without application or library source and without requiring any specially compiled or linked objects. The only requirement is that the application must be dynamically linked to the shared libraries of interest.

Some of these advantages can be found in other tools or other tools can be altered to produce similar results, but SLI pulls it all together in an easily maintainable, dynamically controllable, and customizable package which remains independent of the application and library sources.

3.2. Comparison To Other Techniques

The *trace* command [4,5] of BSD UNIX and *truss* command [6,7] of SVr4 UNIX demonstrate some desirable features. These commands run an application or attach to an active process providing a trace of the system calls made by the application, showing the parameter values and return values or a summary count of all system calls and total time spent in each. *Truss* also allows restrictions on which calls are reported. No special flags are required when the application is compiled. With the exception of attaching to an active process, interposition of dynamic libraries allows all of these features to be applied to user level libraries. Additionally, our tools allow programmatic and interactive control of the data collection and multiple process data collection to a single file or per-process separate files for custom postprocessing reports. For some libraries, such as graphics or database libraries, all updates must pass through the library so parameter and return value capturing is sufficient to record and replay

an application run. This has many positive ramifications on the project lifecycle as discussed in section 6.

System call interposition can significantly expand operating system functionality and transparently provide a number of new services to applications. The COLA [17] and nDFS [18] projects are examples of expanding file system functionality through system call interposition. The Interposition Agents Toolkit [16] presents a number of clever examples of system call interposition utilities and provides an environment to easily create new utilities.

Trace, *Truss*, and *Interposition Agents* are implemented through an operating system trap mechanism for system calls. The trap mechanism allows both dynamically and statically linked applications to benefit from the utilities and allows attaching to an active process. However, the trap mechanism incurs a heavier overhead and is not available for user level library functions. Inversely, interposing on dynamic libraries does not work for statically linked applications and must be selected prior to the application execution but introduces less overhead and allows more libraries to be interposed. Interestingly, the end results of any of these tools and utilities is independent of the interposing technique so informed decisions can be made in the selection of a technique.

Unfortunately, most tools for profiling and tracing require that the source code of the application and libraries be compiled and linked with options different from those of the release executable. That alone can change the executable enough to skew results significantly. The *prof* [4,5,6,7] command of UNIX and its variants *gprof* and *lprof* are the most common profile report generators in the UNIX environment. Special compilation flags cause code to be added to each function to maintain counts. When the application is run, it is interrupted on regular intervals and information about the currently active function is collected. This information, and counts of all function entries, are written to a file upon application completion. This technique has extremely low overhead but lacks detailed accuracy, is limited to the functions that were specially compiled, and only allows a single application per data file. Interposing on dynamic libraries overcomes these restrictions but only for library functions, not the functions of the application. The time interval between library calls can be monitored, giving some measure of application time versus library time, but no detailed profile of the application functions is collected. It is possible to add calls to our toolkit library directly in the application or library source or object but that defeats the concept of interposition to avoid altering application source code and linking.

More onerous than the potentially skewed results of special compilation is the requirement for source code, or at least specially compiled but unlinked object files. Operating system vendors often provide multiple versions of a library: an optimized dynamic version; an optimized static version; a profile static version; and internally there may be debug dynamic and static versions. The application writer then compiles and links appropriately. Debuggers such as *adb* [4,5,6,7], *dbx* [4,5,7], and *sdb* [6] require the source files be present to exploit their full power. Special interposing functions generally require relinking the application with new versions of the functions. It is extremely common to see alternate versions of the libc memory allocation routines *malloc/free* [4,5,6,7,8,9]. The precedence of linking allows interposition to occur either at compile/link time or run-time. Linkers resolve references on a first-come first-serve basis. If an application is linked with two libraries containing functions with the same name, the functions of the first library scanned are used. At compilation/link time, the order of the libraries listed on the link command specify the precedence. At run time, there are a number of environment variables that can alter the order and list of libraries scanned. See section 4.1 for our choice of technique. Some tools such as *Purify*, *Quantify* [9], and *Sentinel* [8] may alter the code contained in the application and libraries or link in new functions not previously included in the application. Relinking requires a new version to replace the function of the original library. The default behavior of our tool is to have a wrapper call the actual function which the application would have used, maintaining precise timing measurement and accuracy with the results of the function call. However, there is no requirement that our wrapper has to call the real routine. The same linker tricks we employ can be used to totally replace the real function or augment with new functionality as in [16,17,18]. More often, we still call the real function but output additional data such as hardware simulation streams.

The granularity of our library tracing technique is limited to the function level. Some profilers such as *tcov* and *Quantify* track hot spots of source code within functions. For a theoretical treatment of hot spot profiling and tracing with source code, see [14]. We consider the function level granularity quite sufficient for our needs, especially when combined with parameter and return value tracing. Lack of access to source code was considered part of our constraints.

4. The SLI Toolset

Tools are provided for varying levels of expertise. Our primary customer is interested in the same li-

ibraries that our group is and utilizes the interposing libraries we've already built and the report generators we've already written. The second tier user needs more or different information than we are providing by default and modifies the interposing library source or the postprocessing report generators to their own needs. Third tier users want to interpose on an entirely new library and so must create their own interposing library from scratch. SLI includes a collection of source, binaries, *awk*, and *perl* scripts to assist and serve as examples to help each level of user.

4.1. SLI Data Collection

Data is optionally collected to three locations. First, cumulative information is kept in shared memory. This cumulative information includes a count of function invocations, how much time was spent in the function, and how much time was spent in the function plus any descendants it may have invoked. Second, data can be written to standard-out or standard-error (stdout, stderr) providing trace and parameter information similar to the output of *truss* or customized output. Third, trace data can be collected to disk including the process-id (pid), the library, the function, the nesting level, how much time has elapsed since the last call into the library, how much time the call took, how much time SLI added in overhead, and parameter values and/or other interesting data. Multiple libraries from multiple applications can write to the same file or each process can create its own file.

The data collection can be controlled programmatically or through a terminal command line interface or through a graphical user interface (GUI). While we consider it desirable to be able to control data collection through scripts and without requiring the window system to be running, experience has shown 100% of the user base uses the GUI ignoring the other two methods. Perhaps this is a skewed sampling since our primary customers are graphics library users.

Data control includes clearing the cumulative information; starting and stopping the stderr output; and starting collection to disk through appending to

existing data or rewinding/truncating the file and starting new. Additionally, data reduction can be controlled to reduce disk overhead; per-library flags can be controlled to alter wrapper functionality on the fly; and inner library nesting call tracing can be controlled (e.g. if the application calls a function in the library and that library function in turn calls another function in the same library, does the user want to capture that "inner library" call or only see what was directly invoked via the application). The figure below is a snapshot of the GUI which controls data collection.

The *ldd* [5,6,7] command lists the dynamic libraries used by a program. The list is generated at compile/link time but the path to locate the libraries can be altered at run-time. Following is an example of the *ldd* output of the *xterm* program (a terminal emulator program in the X-windows environment):

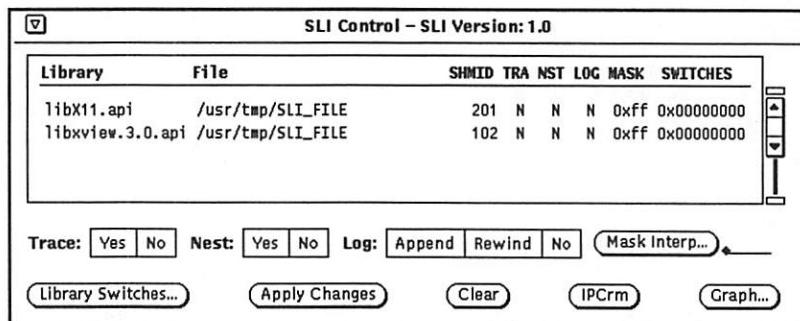
```
% ldd xterm
libXaw.so.5 =>/usr/openwin/lib/libXaw.so.5
libXmu.so.4 =>/usr/openwin/lib/libXmu.so.4
libXt.so.4 =>/usr/openwin/lib/libXt.so.4
libX11.so.4 =>/usr/openwin/lib/libX11.so.4
libdl.so.1 =>/usr/lib/libdl.so.1
libc.so.1 =>/usr/lib/libc.so.1
```

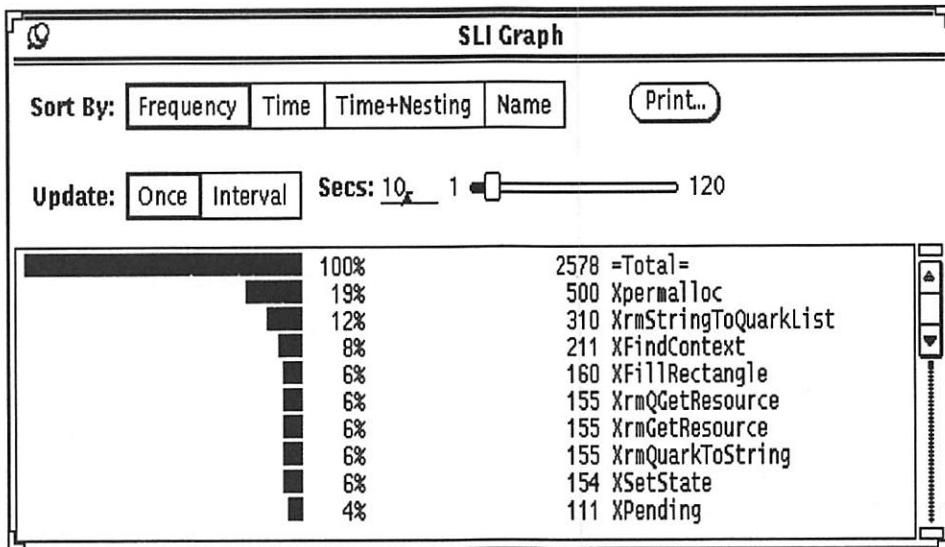
Each of these libraries can have all or any subset of their functions wrapped with data-collecting interposing functions. Furthermore, since *xterm* is an X-windows client application, it is possible to simultaneously profile the X11 server process and correlate interactions between the client and server.

The wrapper functions are invoked through the use of the *LD_PRELOAD* environment variable of the loader. The following example shows how the *ldd* output for *xterm* is changed by this environment variable.

```
% LD_PRELOAD=".:/libX11.api.so ./libsli.so"
% export LD_PRELOAD
% ldd xterm
./libX11.api.so=> ./libX11.api.so
./libsli.so => ./libsli.so
libXaw.so.5 => /usr/openwin/lib/libXaw.so.5
libXmu.so.4 => /usr/openwin/lib/libXmu.so.4
libXt.so.4 => /usr/openwin/lib/libXt.so.4
libX11.so.4 => /usr/openwin/lib/libX11.so.4
libdl.so.1 => /usr/lib/libdl.so.1
libc.so.1 => /usr/lib/libc.so.1
```

You will note that there are now two versions of libX11 listed, our wrapper version and the system version. We've added *api* to the name because we have





three interposing versions of libX11. One version contains the Application Programmer Interface (API) functions where we monitor only those functions the application programmer has access to. A second version includes all functions contained in the libX11 source. A third version interesting to our group contains all the X11 functions with a graphics context parameter. Every symbol we've included in our wrapper library is resolved to us and every symbol we haven't included is resolved through the normal path. The additional library *libsli.so* contains SLI wrapper support functions.

4.2. SLI Reports

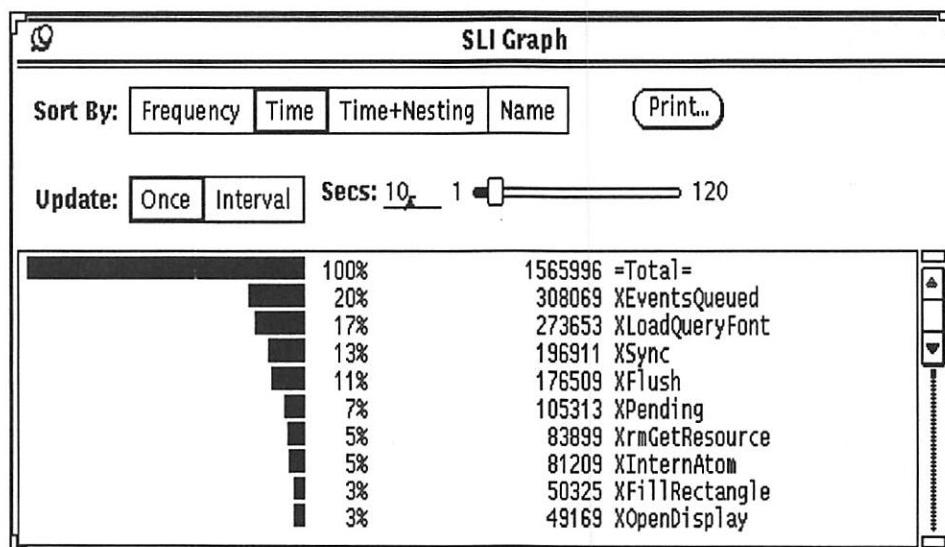
Default reports are provided on the cumulative data and the data written to disk. While data collection is generic, interesting reports on the collected data can be very library specific. Once the data has been collected, it is often necessary to write a custom postpro-

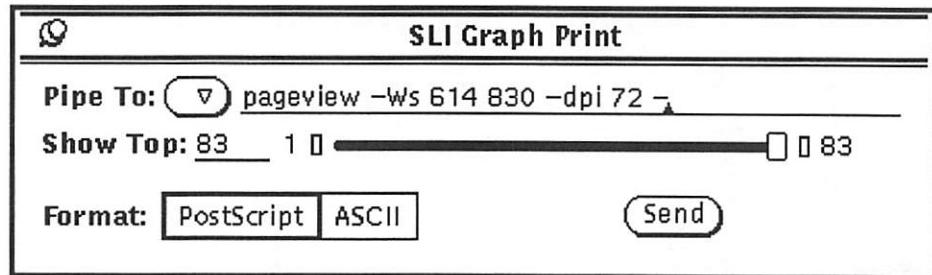
cessing script in *sed*, *awk* [4,5,6,7], or *perl* [15] to glean the interesting information. We provide some postprocessor filters for our graphics libraries which also serve as examples.

The figure above shows a graph of the cumulative information collected on the start-up of an *xterm* before any character has been typed.

From this we note that 2578 calls were made into libX11 of which 500 calls (19%) were Xpermalloc. However, the function called the most doesn't necessarily consume the most time, as demonstrated in the figure below.

The times are displayed in microseconds. This tells us it took slightly over 1.5 seconds to get the *xterm* started. Even though Xpermalloc was the most frequently called function, it is not one of the top 9 consumers of total time.





The graph can be updated on regular intervals while the program is running, monitoring changes during execution. The graph could also be collecting the libX11 calls for all active applications in addition to the single *xterm*.

Two other sorts are provided by default. The time a function took plus all of the interposed descendants it invoked is sometimes more illuminating than the overhead of just the function itself. Second, you may want to find the call frequency or time for a specific function, and the sort-by-name simplifies finding the function.

The cumulative information can be processed through the print option. The default destination is a postscript preview program, but any command can be given to direct the output to a file, printer, or filter program. Generally, the list of interesting functions falls off very fast, so a threshold can be set on how many functions are reported. The format can be in ASCII, allowing postprocessing by custom filter programs.

The data collected to disk is kept in binary form to attempt to reduce the size of the files. A pro-

gram called *sli_interp* is provided to convert the data to ASCII. A postprocessing program can then take that ASCII stream and generate interesting reports. The figure below shows some sample output from *sli_interp*.

The first column provides feedback for nesting information. If no function calls are nested, all of the information associated with a function is kept to one line starting with a vertical bar. If other function calls are made within a traced function, an open brace denotes the entry of a function and a close brace denotes exit from the function. A dashed field means this information can not be provided until the function exits, or it has already been provided on the function entry. This example starts with the application making a call to *xgl_inquire*. The *xgl_inquire* function, in turn, makes several libc calls. We also see that *calloc* makes two other libc calls, *malloc* and *bzero*. The *PID* column traces which process made the call. The *Library* column shows the library, and the *Function* column shows which function of that library. *Nest* shows detailed scope for which functions called which. *Appl* shows how much time passed since the last application call into this library. *Elapsed* shows how much time

Logfile version: SLI log file version 1.0						
X	PID	Library	Function	Nest	Appl	Elapsed
	{	541 libxgl.2.0.api	xgl_inquire	0	1600	----
		541 libc.4.api	malloc	0	0	24
		541 libc.4.api	strcpy	0	11	10
		541 libc.4.api	getenv	0	9	65
		541 libc.4.api	getrlimit	0	1535	61
		541 libc.4.api	malloc	0	9	18
		541 libc.4.api	memset	0	119	24
		541 libc.4.api	strlen	0	1288	8
		541 libc.4.api	writenv	0	312	80090
		541 libc.4.api	read	0	462	1492
	{	541 libc.4.api	calloc	0	38	----
		541 libc.4.api	malloc	1	0	29
		541 libc.4.api	bzero	1	9	12
	}	541 -----	-----	---	-----	70
		541 libc.4.api	ioctl	0	780	108
		541 libc.4.api	getpagesize	0	631	39
		541 libc.4.api	mmap	0	8	333
		541 libc.4.api	mmap	0	497	687
		541 libc.4.api	munmap	0	288	448
		541 libc.4.api	munmap	0	16	558
		541 libc.4.api	close	0	34	81
	}	541 -----	-----	---	-----	307711
	{	541 libc.4.api	sscanf	0	839	----
		541 libc.4.api	strlen	1	0	10
		541 libc.4.api	strcpy	1	10	11
		541 libc.4.api	ungetc	1	109	11
		541 libc.4.api	_filbuf	1	24	8
	}	541 -----	-----	---	-----	205
		541 libxgl.2.0.api	xgl_object_set	0	2864	22
						314 S 3

Function	Blocks	Calls	Prims	Segments	C/B	P/C	S/P
xgl_multimarker	0	0	0	0	0	0	0
xgl_multipolyline	3982	6121	15981	69715	1.537	2.610	4.362
xgl_multi_simple_pol	0	0	0	0	0	0	0
xgl_polygon	607	607	607	1214	1	1	2
xgl_triangle_strip	53	673	673	60174	12.69	1	89.41
xgl_quadrilateral_me	0	0	0	0	0	0	0
xgl_stroke_text	0	0	0	0	0	0	0
xgl_annotation_text	0	0	0	0	0	0	0
Totals:							
Markers				0			
Lines				69715			
Chars				0			
Triangles				61388			
Calls to xgl_context_post()				4374			
Calls to xgl_context_new_frame()				124			
Calls to xgl_object_get()				124			
Timing:							
appl+func+sli time:				134.20			
sli time:	6.14			5%			
appl+func time:		128.06		95%			
appl time:	18.77	15%		14%			
func time:	109.28	85%		81%			

was spent on this function call. The *SLI* column shows how much time overhead SLI introduced to collect this information. *Data* optionally contains parameter or other interesting data values before and after the function was invoked.

All of these fields can optionally be omitted from the data collection in order to do up-front data reduction. If the user knows only one process is being profiled or only one library is being traced, then the user can select not to save those fields in the binary file. The *sli_interp* program knows how to handle the reduced data files.

Library specific postprocessors can produce useful reports from the *sli_interp* output. For example, the report in the figure above is a summary of what graphics primitives were used during an application run, how well the application merged multiple primitives into a single library call, and breaks out the percentages of time spent in the application versus the library versus the overhead introduced by SLI. In this example, the entire run took just over 2 minutes (appl + func + sli time = 134.20 seconds). Only 14% of the time was spent in the application and 81% of the time was spent in rendering the graphics. The overhead introduced by SLI only represented 5% of the total time (6.14 seconds). It is fairly obvious that the shared memory data produces the lowest overhead, but not so obvious that the binary data collected to disk is much faster than the formatted ASCII output sent to stderr. SLI memory maps the file and accesses it as if it were memory, leaving it up to the system to write it back to disk when necessary. Contrast that to a formatted print statement being output to a scrolling terminal and the reason for the different overheads becomes more clear.

4.3. Interposing Library Source

The second and third tier users want to go beyond the default reports and libraries that our group has provided. This implies they need to alter the interposing libraries we provide or create new interposing libraries. We provide the source to each interposing library we've already created and we also provide tools to automate the process of creating a new interposing library. There are several steps to creating a new interposing library that have taken us as little as 20 minutes for one library and as long as two weeks for a particularly difficult library whose default output format just wasn't what we wanted to report to our users. On an average, most of our customers have been able to get a useful interposing library in one half-day of effort.

The first step is to create what we call a *prototype* file. This is a file that consists of the function declarations for all functions to be traced. Generation of this file is generally quite easy. For an Application Programmer's Interface library, the declarations are already in a header file. Lint library declarations can serve as a source as well. If C program source is available (either K&R-C [11] or ANSI-C [12]), then the *cproto* program [13] quickly and easily generates the prototype file. Using *cproto* has been our primary method. C++ turns out to be quite difficult to generate interposing libraries for and libc provides some special challenges. Both of these are discussed in more detail in the "Obstacles" section.

Since libc is quite common to many operating systems, we will use a small example of *calloc* and follow it through to an interposing library. The prototype file would contain:

```
void *calloc( size_t num, size_t size);
```

We allow single line comments and preprocessor directives to be placed in the prototype file as well. These are passed directly from the prototype file to the generated C code. The prototype file is processed by an awk script that generates two files. One file is a “translation file” that tracks the total number of functions in the library, the length of the longest function name, and a translation table from a numeric assignment to the ASCII name of the function. Since this information is static once the library is generated, it aids the dynamic allocation of arrays during the data collection phase of profiling. The number-to-name mapping allows compact information to be written to the data file in binary. The second file from the awk script is the C source for the interposing library. We call this a working wrapper template. We use the term *working* because the generated code can compile and be useful right away, but we add the term *template* because truly interesting, detailed data collection generally requires customization of the generated code. Knowing the name and size of parameters is useful, but contextually understanding the contents of a complex structure and what’s interesting generally requires human intervention and customization. The generated C code for the `calloc` prototype is:

```
void *calloc( size_t num, size_t size)
{
    static char *func_name = "calloc";
    typedef void *(*real_func_type)
        ( size_t num, size_t size);
    static real_func_type real_func;
    void *return_value;
    int save_sli_active = sli_active;

    SLI_DECLARE

    if (sli_active)
    {
        if (!real_func)
            real_func = (real_func_type)
                (*sli_resolve(3, func_name));
        return((*real_func)(num, size));
    }

    sli_mark(SLI_MARK_SLI_ENTER);
    sli_active = 1;
    if (!sli_lib_info_3)
        sli_lib_info_3 = sli_find_info(3);
    if (sli_lib_info_3->tra_ctl == SLI_TRA)
        sprintf(SLI_STDOUT,
            "calloc( num=0x%0x size=0x%0x)\n",
            num, size);
    if (!real_func)
        real_func = (real_func_type)
            (*sli_resolve(3, func_name));
    SLI_PROLOG
    sli_send(SLI_ENTER, 3, 15, SLI_EOP);
    sli_active = 0;
    sli_mark(SLI_MARK_FUNC_ENTER);
    return_value = (*real_func)( num, size);
    sli_mark(SLI_MARK_FUNC_EXIT);
    sli_active = 1;
    sli_send(SLI_EXIT, 3, 15, SLI_EOP);
    SLI_EPILOG
    sli_active = save_sli_active;
    sli_mark(SLI_MARK_SLI_EXIT);
    return return_value;
}
```

This example serves to illustrate several points. First and foremost, strong typing must be followed for the return types and parameter types. Different compilers have different rules for data type sizes, calling conventions, and parameter promotion rules. The template is careful to cast all types. This template is for an ANSI-C compiler. The same awk script knows how to generate output for K&R-C and C++ with minor variations. The function name is placed in a variable per function so it can be symbolically referenced in the `SLI_DECLARE`, `SLI_PROLOG`, and `SLI_EPILOG` macros. These macros are null by default but provide a means for all functions to have common code added with ease.

You will see the constants 3 and 15 throughout the template. Both of these constants are the number-to-name mapping values generated in the translation file. The 3 is for `libc` and the 15 is for `calloc` in `libc`.

Since this function is in `libc`, there is some added code that is not typically found in the templates. The initial check for the global variable `sli_active` is a hook to avoid recursing on `libc` from our interposing code (that is to say, we want to trap `libc` calls from the application and the functions we are tracing but we don’t want to trap `libc` calls that our tracing software uses). A global variable lacks elegance but provided a quick solution to trace all `libc` functions. A better solution will be required to support multiple threads.

The `sli_mark` function is used to track the time durations of the overhead SLI has introduced and the duration of the real function when it is called. There are four `sli_mark` calls. First on entry to the wrapper, second just before the real function is called, third immediately upon return from the real function and fourth on exit from the wrapper. For non-`libc` wrappers, `sli_mark` is the first executable statement.

The first time any function is called in the library, some initial one-time overhead is incurred. The `sli_lib_info` is a shared memory page that is used for multiprocessing locking and run-time interactive control of profile and trace functionality. Likewise, the first time a wrapper function is called, we have to find the pointer to the real function. This pointer is saved so the overhead is only encountered once per function.

The `tra_ctl` structure member contains the current value for the “trace to stderr” option. This is the only data collection under complete control of the customizing user. The shared memory data collection and collection to the file is handled by the support library through the `sli_send` function. A variable list of parameters can be sent and stored as the data field in the binary file.

This is as much as can be automated without contextual knowledge associated with the functions. For instance, the parameters are always just printed as a hex value. Often, a parameter might be a complex structure requiring human intervention to know what information in that structure is interesting and what format it should be printed in. Similarly, it would be inappropriate to simply save all parameter values to the trace file causing a tremendous growth in size. It is more appropriate not to save the parameters by default and allow human intervention to decide what information is important to save and in what format.

5. Obstacles Encountered

There are a number of issues that get in the way of implementing an interposing library. Fortunately, nearly all are solved, although some require fairly detailed system knowledge.

5.1. Finding The Real Function

This is potentially difficult to figure out for an arbitrary operating system. The Solaris 2.3 operating system provides a simple interface to accomplish this. The *dlsym* function is used to find the address of a symbol in a dynamically linked library. Solaris 2.3 provides a special parameter to *dlsym* called RTLD_NEXT which indicates to “find the next address of this symbol in the list of libraries”. That’s all it takes. For standard SVr4 and earlier versions of Solaris, *dlsym* is available but does not support RTLD_NEXT. The solution we followed is to traverse the linker structures and locate the list of libraries through them. We then *dlopen* each library in the list and loop through the list with *dlsym* looking for the real function. This is not too difficult and is somewhat documented [1] but should not be considered a normal, supported user interface. There are two concerns with this approach. One, be careful not to find your own symbol and get caught in a recursive loop. Two, for the sake of efficiency, you only want to loop through the libraries once to find the first function used in a library and keep that handle around for subsequent symbols from the same library.

It is not uncommon for compilers to slightly alter the names of functions from the source to the object file. This usually takes the form of an underscore character placed at the front and/or back of the name. The *dlsym* function properly handles the underscore for the programmer. The C++ language adds considerably more information including the class and parameter type information in the object file function name. This presents a problem in specifying the name of the real function to *dlsym*. For C++ compilers that preprocess

the C++ code into C and then use the C compiler, you can collect the “mangled” name from the C code. For C++ compilers that compile directly into object files, you may need to use the *nm* [4,5,6,7] command to determine the mangled names.

5.2. C++

There are a number of interesting problems that arise when attempting to interpose on C++. As just mentioned above, finding the real function with the name mangling schemes is one hurdle. Generating the list of function prototypes can be much more complex than with C. Interposing on a programmer’s interface is still straightforward, but finding all of the functions in a C++ library can be quite difficult. A proper profile and trace of a library needs to know where all of the overhead comes from. Scanning the source is insufficient since C++ may generate a lot of functions for the programmer. Examples of generated functions are constructors, destructors, copy operators, function templates, and virtual functions. The solution appears to be to query the library object files for functions and reverse that back to function prototype declarations. However, this can still be missing critical information such as default parameter values and full type information and class member function declarations for compiler created functions. At this time, C++ still remains a challenge which we’ve only partially solved.

5.3. Interposing On libc

Our support library is written in C and uses many functions from libc. Furthermore, most of the libraries we interpose on are also written in C and make use of libc. When we finally decided to add libc to the list of supported libraries, we found ourselves with recursive looping problems. The COLA project [17] also uses LD_PRELOAD to interpose on system calls and reports a similar looping problem.

Our solution required two steps. First, the interposing version of libc checks a global variable to know if it is being called from an interposing or SLI internal function rather than an application or regular library function. If so, it calls the real function directly without collecting any statistics. As noted in section 4.3, the use of a global variable will prove to be a problem when we want to support multiple threads. This is our only global variable and might be fixed by making it a thread specific variable. Second, the routine to find the real function had to be made “libc clean”. That is to say, it couldn’t have any references to libc in that one function or it had to precisely resolve any libc functions it did use directly to the real libc library.

5.4. LD_PRELOAD Side Effects

LD_PRELOAD should be used with care. One side effect of this environment variable is that the interposing functions are loaded for all commands issued. You may end up collecting data from more processes than expected. Also, if the interposing functions reference symbols expected to be resolved by libraries of the application, other commands might not have included those libraries, leaving those symbols unresolved causing command execution failure. This can be overcome by linking each interposing library to the library it interposes.

5.5. Scoping Issues

All functions in a library made available to an application programmer have to be declared global. However, it is possible that the library may have internal support routines that it uses but does not expose to the application programmer. If a function is declared static, then it can not be interposed. We generate multiple versions of interposing libraries for each target library. One consists exclusively of the functions available through the Application Programmer Interface (API); a second for all non-static routines in the source; and sometimes a third containing a specifically interesting subset of functions.

Global variable references can be a problem if not considered carefully. Two functions within a library may share access to a global variable. The scope of that global variable and whether or not the interposing functions are interested in that variable raises some issues. If the variable is global to the entire library and application, then no problem exists. If the variable is shared between two functions in the same source file but not global to the library and application, then it may not be accessible. Similarly, we have encountered some compiler discrepancies on inner library calls. If two functions *foo* and *bar* are contained in the same source file, and compiled to the same object file, and *foo* calls *bar*, some linkers improperly resolve *bar*'s reference in *foo* at link time rather than run-time which prevents interposition. This is actually a bug and if encountered, can generally be overcome through compiler or linker command line options.

5.6. Parameter Handling

It is reasonable to believe that a function found in a library is independent of the compiler it was generated from, but in reality, problems such as parameter promotion and variable parameter list handling can present particularly difficult problems to isolate and resolve.

A hard and fast rule to apply is: use the same compiler for your interposing function as the real library. K&R-C compilers [11] have different parameter promotion rules from the ANSI-C standard compilers [12]. If the interposing function does not properly pass the parameters down to the real function or properly pass the return value back to the application, the interposing function is useless.

Variable parameter list functions are an especially interesting problem to solve generically since it is the responsibility of the called routine to determine how much information to read from the stack. The interposing function has the responsibility to pass the correct amount of information down to the real function. We solved this two ways. One, for our automatically generated interposing functions that contain variable argument lists, we have some in-line assembly routines inserted that copy the entire frame of the calling routine into the stack for the real function. This can potentially copy too much information, generating unnecessary overhead, but guarantees the real function receives everything. The other solution is to customize the interposing routine to know how to parse the stack and pass down the correct amount of information. This too adds overhead since the stack must be both parsed and copied, but insures only the necessary amount of information is copied.

5.7. Multiple Processes and Processors, Threads, and Network Implications

Data collection and interpretation is straightforward if only one process is collecting data, but multiple process data collection is too valuable to ignore. For example, setting LD_PRELOAD to include all graphics libraries before starting the window system allows capture of all frame buffer activity for every application. The three data collection points plus the central control area must maintain atomic transactions for updates. For multiple processes on multiple processors on a single system, this can be handled fairly easily through an atomic read-modify-write semaphore in shared memory. Multiple threads of the same process on multiple processors adds complications. The same process-id may have mixed library function entry and exit flows. A thread identification needs to be included with the process-id to sort out data flow and maintain nesting stacks. Multiple processes running on separate machines in a network are very difficult to synchronize and we are only starting to tackle that problem.

Semaphore locking adds the potential for deadlocks. We only lock when we are ready to update shared information and then immediately free the lock.

This has only been a problem when the application being profiled is killed. Our solution is to have the request for a lock time-out and clear the lock itself, reporting that the data "may be corrupted".

5.8. Timer Overhead

The resolution of the timer can make a major difference in the usefulness of a profiler. Initially, we used the *gettimeofday* libc function, but found the overhead of a regular system call took on the order of 50 milliseconds when we wanted resolution on the order of nanoseconds. Under Solaris 1, we wrote a device driver to provide direct user reads of the system clock. Under Solaris 2, a new function *gethrtime* is provided. Both of these gave us around 2 microsecond clock resolution improving our accuracy considerably.

6. Application of the Tools

The tools have proven quite successful in quickly isolating performance bottlenecks in the use of graphics libraries and have provided the expected feedback to both the application writer and the library writer. What was unanticipated was the amount of information we could gather and how that information could be applied.

First, once the tools were in place, we found adding new libraries to be trivial (with the single exception of libc). In one case, the time between the request for a new interposing library and the time it was ready was only twenty minutes. In general, we are now surprised if it takes us more than two days to overcome any difficulties in creating a new interposing library. We originally anticipated looking at only a few libraries. The ease of adding new libraries has led to a quick proliferation of new libraries on demand and spread beyond graphics libraries. Sun customers who were shown the tool to assist in graphics performance have taken the initiative to create interposing wrappers for their own libraries.

The application developer uses the default reports and the postprocessing reports to be able to make better use of the library. The hardware and library developers get feedback on actual usage patterns in the library. That information can be applied in many ways. The library builder can sort functions that are often used together to provide cleaner paging. Application regression tests can show what primitives and attributes are used and which aren't (and thus candidates for eventual removal). Analysis of benchmarks, demonstrations, and actual application usage emphasize what functions are critical and with what attributes or parameter values, what functions are time consuming,

and what functions deserve the most attention and possible hardware acceleration.

The ability to capture all of the calls and parameters has potential beyond simple playback. If an application records a session, encounters a bug, and that bug reproduces in the playback, then the odds are pretty good that the bug really is in the library or bad parameter values were passed to the library by the application. Either way, vendor support and bug reporting can reproduce and analyze the bug without having to acquire the application, data, and instructions. Additionally, the playback program (or even the wrapper) need not actually call the real library but can instead be used as a translator. The translator might emit simulation traces allowing developing hardware to test different schemes against real application data patterns. The translator might emit calls to an alternate or new version of the library testing the robustness and performance of the new library before the application has actually been ported. Furthermore, the playback is often considerably faster than the original application since the computations leading to the function calls are already made. This means that bug tracing is much quicker and easier. Additionally, having the source to the playback program rather than the original application means special interposers like Purify can be applied to the run by recompiling and linking the playback program rather than the application source. If the wrappers are compiled with debug flags, then a debugger can be used to provide functionality on library functions you wouldn't normally have access to. An example is conditional breakpoints based on contextual parameter values to obtain a callback stack on a system call.

7. Conclusions

Dynamic library interposition has been extremely successful for us. We have been able to exploit the detailed information in many different and useful ways for many different libraries. The value of tracing the parameters in addition to the functions, should not be underestimated. Initially developing the tools was nontrivial but with the tools in place, our development teams are able to make much more informed decisions based on real workloads and fewer guesses. We've generated approximately 40 interposing libraries of both Sun supplied libraries and third party libraries. Around a dozen applications have used SLI as the primary analysis tool with significantly improved performance. Playback to test pre-release hardware and software improved release quality, and hardware simulation trace files are currently being generated for projects in progress.

Acknowledgments

Recognition must go to Doug Gehringer and Dave Phillips for being the first users, major shapers in the directions of the development, and contributors to the playback, symbol resolution, and early timer code. Thanks to Rob Gingell and Rod Evans for quick fixes and knowledge dumps on the strange linker magic we've encountered. Timothy Foley has been our front line and primary evangelist for SLI, and lots of support and patience can be credited to Dean Stanton. Additionally, Brian Herzog, Ralph Nichols, Matt Perez, Jon Cooke, Mike Penick and Roger Day all had the foresight to take the long term view and support tool development despite the difficulty up front in demonstrating the difference tools can make to the bottom line.

References

- [1] "System V Application Binary Interface", UNIX Press, Prentice-Hall Inc., 1990, ISBN 0-13-877598-2.
- [2] "SunOS 5.3 Linker and Libraries Manual", SunSoft, Part No: 801-5300-10, November, 1993.
- [3] Gingell, R. A., M. Lee, X. T. Dang, M. S. Weeks, "Shared Libraries in SunOS", Summer Conference Proceedings, Phoenix, 1987, USENIX Association., 1987.
- [4] "Unix User's Manual: Reference Guide", 4.2 Berkeley Software Distribution, March, 1984.
- [5] "SunOS Reference Manual", SunSoft, Mountain View Ca., Part No: 800-3827-10, March, 1990.
- [6] "System V Interface Definition", AT&T, Third Edition, 1989.
- [7] "SunOS 5.3 Reference Manual", SunSoft, Mountain View Ca., Part No: 801-5297-10, October, 1993.
- [8] "The SENTINEL Debugging Environment", Virtual Technologies, Inc., Dulles, VA. info@vti.com.
- [9] "Purify User's Guide", Pure Software Inc., Sunnyvale, CA. info@pure.com.
- [10] "Introduction to SPARCworks", SunPro, Mountain View CA, Part No: 800-7262-11, October, 1992.
- [11] Kernighan, Brian W., Dennis M. Ritchie, "The C Programming Language", Prentice-Hall Inc., 1978. ISBN 0-110163-3.
- [12] Arnold, Ken, John Peyton, "A C User's Guide to ANSI C", Addison-Wesley Publishing Co., 1992. ISBN 0-201-56331-2.
- [13] Huang, Chin, "Cproto Manual", cthuang@zerosan.uucp or chin.huang@canrem.com, 1993.
- [14] Ball, Thomas, James R. Larus, "Optimally Profiling and Tracing Programs", ACM, 089791-453-8/92/001/0059, 1992.
- [15] Wall, Larry, Randal L. Schwartz, "Programming perl", O'Reilly & Associates, Inc. 1990.
- [16] Jones, Michael B., "Interposition Agents: Transparently Interposing User Code at the System Interface", Proceedings of the 14th ACM Symposium on Operating Systems Principles. Asheville, NC, December, 1993.
- [17] Krell, Eduardo and Balachander Krishnamurthy, "COLA: Customized Overlaying", Winter USENIX Conference Proceedings, January, 1992.
- [18] Fowler, Glenn, Yennun Huang, David Korn, Herman Rao, "A User-Level Replicated File System", Summer USENIX Conference Proceedings, June, 1993.

Tim Curry is a Senior Staff Engineer with Sun Microsystems. He is currently working in the Technology Development group of Sun designing portable workstation hardware. He has been with Sun since 1985 primarily in windows and graphics software. Tim has a B.S., M.S., and Ph.D. in Computer Science from the University of Central Florida.

Large Granularity Cache Coherence for Intermittent Connectivity

L. Mummert, M. Satyanarayanan
Carnegie Mellon University

Abstract

To function in mobile computing environments, distributed file systems must cope with networks that are slow, intermittent, or both. Intermittence vitiates the effectiveness of callback-based cache coherence schemes in reducing client-server communication, because clients must validate files when connections are reestablished. In this paper we show how maintaining *cache coherence at a large granularity* alleviates this problem. We report on the implementation and performance of large granularity cache coherence for the Coda File System. Our measurements confirm the value of this technique. At 9.6 Kbps, this technique takes only 4 – 20% of the time required by two other strategies to validate the cache for a sample of Coda users. Even at this speed, the network is effectively eliminated as the bottleneck for cache validation.

1 Introduction

Callback-based cache coherence [4, 10] in distributed file systems has proven to be invaluable for preserving scalability while maintaining a high degree of consistency. This technique is based on the implicit assumption that the network is fast and reliable. Unfortunately, this assumption is often violated in mobile computing environments. Network communication in those environments is often slow and intermittent.

Instead of requiring a client to check the validity of a file on each access, a callback-based scheme places

This research has been supported by the Advanced Research Projects Agency (Hanscom Air Force Base under Contract F19628-93-C-0193, ARPA Order No. A700), IBM Corporation, Digital Equipment Corporation, Intel Corporation, and Bellcore. The views and conclusion expressed in this paper are those of the authors, and should not be interpreted as those of the funding organizations or Carnegie Mellon University.

Authors' addresses: School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA, 15213-3891. e-mail: lily@cs.cmu.edu, satya@cs.cmu.edu.

greater responsibility on the server. When a client caches a file from a server, the server promises to notify it if the file changes. This promise is called a *callback*. An invalidation message is called a *callback break*. If a client receives a callback break for a file, it discards the cached copy and re-fetches it when it is next referenced.

When a client with callbacks encounters a network failure, it must consider its cached files suspect because it can no longer depend on the server to notify it of updates. Upon repair of the failure, the client must validate cached files before use. Consequently, as failures become more frequent, the effectiveness of a callback-based scheme in reducing validation traffic decreases. In the worst case, client behavior may degenerate to contacting the server on every reference. This problem is exacerbated in systems that use anticipatory caching strategies such as *hoarding* to prepare for failures [1, 5]. In these systems, validation traffic is proportional not just to the file *working set*, but to the larger *resident set*. The more diligent the preparation for failures, the larger the resident set. The impact of this problem increases as network bandwidth becomes precious.

We can address this problem without weakening consistency by increasing the *granularity* at which cache coherence is maintained. This makes validation more efficient, allowing clients to recover from failures more quickly. Taken to an extreme, this idea would require maintaining a version stamp and callback on the entire file system. If the version stamp remains unchanged after a failure, the client can be confident that no files have been updated on the server. A callback on the entire file system is a very strong statement — it means every file cached at the client is valid. However a callback break on the file system conveys little information — anything in the file system could have changed, whether cached at the client or not. A practical implementation of this idea requires a choice of

granularity that balances speed of validation with precision of invalidation.

In this paper we report on the implementation and performance evaluation of large granularity cache coherence in the Coda file system [10]. Our results show that large granularity cache coherence is well-suited for a significant fraction of what Coda users typically cache. At 9.6 Kbps, this technique takes only 4 – 20% of the time required by two other strategies to validate the cache. Effectively, this technique eliminates the network as the bottleneck for cache validation. At higher bandwidths, the value of this technique diminishes, but it is always at least as good as the other strategies.

The paper begins by introducing key aspects of Coda. Then we describe the operation of the system with multiple granularities. Details concerning the actual implementation are presented in Section 4. Finally we describe the current status of our system and evaluate its performance.

2 Coda File System

Coda is a descendant of AFS-2¹ [4] that has *high data availability* as its main goal. Like AFS, it provides a single, shared, location-transparent name space, and maintains cache coherence using callbacks. Files are stored in *volumes* [11], each forming a partial subtree of the name space. Volumes are administrative units, typically created for individual users or projects. A user-level process called *Venus* manages a file cache on the local disk of each client. Venus makes requests of servers through the *Vice interface* using remote procedure calls (RPC). Files are identified by *fids*, which are 96 bits long. The first 32 bits of the fid are the volume identifier.

Coda uses two strategies to achieve high data availability: *server replication* and *disconnected operation*. Server replication allows volumes to be stored at a group of servers called the *volume storage group* (VSG). At any time, the subset of those servers available is called the *accessible volume storage group* (AVSG). When making requests, clients contact all servers in the AVSG (though data is fetched from only one), and all servers maintain callbacks for objects cached from the VSG. If an AVSG grows, clients drop callbacks for objects stored at that VSG, because the newly available server may contain more recent data. Further details on server replication may be found in [10].

¹AFS has evolved since the version from which Coda was derived, which was AFS-2. The currently deployed version is AFS-3. Unless qualified, the term AFS applies to both versions.

Disconnected operation arises when the AVSG becomes empty. To prepare for disconnection, users may *hoard* data in the cache by providing a prioritized list of files called the *hoard database*, or HDB. Venus combines hoard database entries with LRU information as in traditional caching schemes to implement a cache management policy that addresses both performance and availability concerns. Periodically, Venus *walks* the cache to ensure that the highest priority items in the HDB are cached and consistent with the AVSG. Hoard walks may also be requested explicitly by the user. If an object in the HDB is invalidated, it is re-fetched on the next reference or during the next hoard walk, whichever comes first. Hoard walks can create bursty network traffic. A hoard walk after an AVSG grows results in a validation request for every cached object from that VSG. More details on hoarding and disconnected operation may be found in [5].

3 Protocol Description

At how many granularities should cache coherence be maintained? In principle there can be many levels. An obvious mapping onto a Unix file name space would suggest a hierarchy of granularities. But the desire for a simple implementation led us to use just two: files² and volumes. Volumes are attractive as units of coherence because they tend to represent groups of files that are logically related and hence possess similar update characteristics.

When a client maintains coherence on files, it must validate them before use when the AVSG has grown. This approach is based on the assumption that the newly available server has rendered some file in the cache stale, necessitating a check of each one. As failures become more frequent, the price of suspicion increases. Increasing the granularity of coherence allows a client to summarize the contents of its cache for the purpose of validation. This approach is more optimistic, in that it assumes there are sets of cached files that have not changed during the failure.

To summarize cache state by volume, servers maintain version stamps for each volume they store. The version stamp for a volume is incremented whenever an object in the volume is updated. A client caches the version stamp, establishing a callback for the volume. When the AVSG grows, the client validates the files in a volume by sending its version stamp to the server. If the stamps match, all of the client's cached data from the volume is valid. The server grants a callback for the volume to allow the client to read the

²In this paper, we use the term *file* to refer to single objects in the file system, including directories and symbolic links.

cached files without any additional communication. If the validation fails the client reverts to file callbacks.

We expect maintaining coherence on volumes to be beneficial for collections owned by the primary user of a client, and for collections that don't change frequently or change *en masse* (e.g., system binaries) [8]. In Section 5, we show that such collections represent a large fraction of the files that users cache. File callbacks are more appropriate for volumes that are shared or owned by users other than the primary user of a client.

Of course, the client must ensure that version stamps are consistent with the data they represent, and it must handle updates from other clients, which manifest themselves as callback breaks. We discuss these issues further in the remainder of this section.

3.1 Obtaining Callbacks

A client caches a volume version stamp to prepare for the next failure. If a client presents an up-to-date stamp after a failure, it is granted a callback on the volume. The volume callback is a substitute for file callbacks on all the files in that volume. The callback is actually on the version stamp. It means the client has files corresponding to the version of the volume designated by the value of the stamp.

Before obtaining a volume version stamp, we require all files in the cache from that volume to be valid and have callbacks. This ensures the files at the client correspond to the version stamp it receives. Since validating the files could be expensive, the client should employ a policy that balances this cost with the expected value of having a volume version stamp in case of a failure. We discuss policy further in Section 4.3. For volume callbacks to be effective, there should be more than one file cached from the volume.

If the client holds a volume callback and fetches a new file, the server establishes a file callback for the new file. This is not necessary for correctness, but it is useful for performance. Although one could imagine not establishing the file callback to conserve server memory, granting the file callback in this case requires no additional network traffic, and gives the client something to fall back on should the volume callback be broken.

3.2 Handling Callback Breaks

When a file is updated by a remote client, the server breaks callbacks to all other clients holding a callback for that file or its volume. If a client holds callbacks on both the file and the volume, the server breaks the

callback on the file. The client interprets this as a callback break on the volume as well, and erases its version stamp. Note that if a client holds a volume callback, it will receive a callback break even if the updated file is not in its cache. This is *false sharing*, and if frequent, may indicate that the granularity of cache coherence is too large for that volume. The client should not blindly reestablish the callback when it is broken, because updates exhibit temporal locality [2, 9]. Not only would this be a waste of bandwidth, but it would also harm scalability. The client's policy should take this into account when determining whether the volume callback should be reestablished.

The presence of both volume and file callbacks means clients must decide what kind of callback to obtain when one is broken. Suppose a client validates a version stamp for a volume, and it receives a volume callback. At this point it has no file callbacks. If the volume callback is broken, the client must validate its cached files from that volume before it can reestablish the volume callback. In terms of network usage, this is equivalent to recovery from a failure without volume callbacks. In effect, the client has delayed validation of individual files.

In the situation above one might imagine obtaining file callbacks in the background in case the volume callback is broken. This eager strategy assumes a remote update will occur before the next failure. However, this defeats the purpose of obtaining a volume callback. Instead, we employ a lazy strategy, obtaining file callbacks only if the volume callback is actually broken. If no remote updates occur between failures, we have saved the network bandwidth and server memory that would have been required to validate and obtain file callbacks.

4 Implementation

We layered volume callbacks on the existing callback mechanism as much as possible. Code changes were required in the Vice interface, the server, and Venus. We discuss these changes in the following subsections.

4.1 Vice Interface

We added two new calls to the Vice interface that manipulate version stamps, which were already being maintained by each server for replication. The first new call is `ViceGetVolVS`, which takes a volume identifier, and returns a version stamp and a flag indicating whether or not a callback has been established for the volume.

```
ViceGetVolVS(IN VolumeId Vid,
    OUT RPC2_Integer VS,
    OUT CallBackStatus CBStatus);
```

The second call, *ViceValidateVols*, takes a list of volume identifiers and version stamps and returns a code for each indicating if it is valid, and if so, whether a callback has been established for the volume. The structure *RPC2_CountedBS* consists of a length field and a sequence of bytes.

```
ViceValidateVols(
    IN ViceVolumeIdStruct Vids[],
    IN RPC2_CountedBS VS,
    OUT RPC2_CountedBS VFlagBS);
```

Besides the two new Vice calls, there are also new parameters to existing calls that perform updates (*mkdir*, *rename*, etc.).

4.2 Server side

Server code is required to support the new Vice RPCs, and volume callbacks themselves. We added about 400 lines of code to the server, which consists of approximately 14,500 lines of code excluding headers and libraries. Most of the changes involved supporting the new RPCs (200 lines) and debugging and printing statistics (150 lines). The remainder of the changes were for gathering statistics.

We minimized changes to data structures and code involving callbacks by designating an unused fid (*<VolumeId>.0.0*) to represent an entire volume. We modified the callback break routine to break callbacks not only for a file, but also for the volume that contains it.

Updates change the volume version stamp, whether they are made remotely, or by a local client. When a client updates a file, it receives a status block containing the file's new version information and attributes. The status block is shown in Figure 1. Similarly, the client must be able to observe the effects of its updates on the volume version stamp, without receiving callback breaks or sending additional messages.

We considered two approaches for updating the client's version stamp when it performs an update – having the client compute the new stamp, or having the server compute and return it. The advantage of having the client compute the new stamp is no additional changes need to be made to the Vice interface. Unfortunately, since the server must maintain version stamps anyway, this approach duplicates a good deal of code, and is more difficult to test and maintain.

We chose to have the server compute and return the new version stamp. We have added three parameters to Vice calls that involve updates:

- the old version stamps
- the new version stamp
- the callback status

When a client performs an update, it sends its copy of the volume version stamp to the server along with the other parameters for the operation. If the client's stamp is current, the server returns the new stamp and a callback for the volume. If it is not, the server returns a zero stamp, and no callback. If the client does not have a stamp, or does not wish to obtain a volume callback, it simply sends a zero stamp. This is guaranteed never to match at the server.

This process is complicated by concurrency control. Files involved in an update are locked for the duration of the operation. For performance reasons, the server cannot lock a volume for the entire duration of an update. Therefore, it is possible for updates to different objects in a volume to be interleaved. To detect this, the server updates the client's version stamp along with its own, and checks for a match at the end of the call.

There are operations other than file updates that change volume version stamps. We made a few additional changes to two server libraries to ensure callbacks would be broken when these operations occurred. One of the libraries supports debugging; the other is part of the resolution subsystem [7].

Our implementation was complicated by a number of race conditions, pertaining to server replication, that manifested themselves during initial testing. These race conditions were present in the original code from which Coda is derived, but were triggered when clients eagerly acquired volume callbacks.

For example, the callback processing code is structured to prevent a server from adding a callback for a fid while breaking a callback for that fid. Callbacks are maintained at all servers in the AVSG. The race condition occurs when a client receives callback breaks from a subset of the AVSG and immediately tries to reestablish its volume callback. This request is sent to all the servers in the AVSG; this may include ones still breaking the callback. This used to cause the servers to crash. We fixed this by returning the callback status, and having the server not establish callbacks in this situation.

```

typedef RPC2_Struct
{
    RPC2_Unsigned    InterfaceVersion;
    ViceDataType     VnodeType;
    RPC2_Integer     LinkCount;
    RPC2_Unsigned    Length;
    FileVersion      DataVersion;
    ViceVersionVector VV;
    Date             Date;
    UserId           Author;
    UserId           Owner;
    CallBackStatus  CallBack;
    Rights           MyAccess;
    Rights           AnyAccess;
    RPC2_Unsigned    Mode;
    VnodeId          vparent;
    Unique           uparent;
} ViceStatus;

```

Figure 1: Vice Status Block

This figure shows the Vice status block, which is returned for the objects of most Vice calls. It includes version information for the object, whether or not the server has extended a callback promise for it, and the access rights of the requesting user and the anonymous user System:Anyuser.

4.3 Client side

Most of the logic for supporting volume callbacks is in Venus. In addition to using the new RPCs, Venus must cope with replication, and decide when using volume callbacks is appropriate. The changes represented an addition of 700 lines to about 36,000 lines of code excluding headers and libraries.

The implementation of Venus is layered with respect to files and volumes. The changes for volume callbacks are concentrated in the volume layer, leaving the heart of Venus unchanged. We augmented the volume data structure to store a volume version stamp, the status of a volume callback, and summary statistics such as the number of callbacks established, broken, and cleared.

There are a number of background processes within Venus that run periodically. The hoard daemon, for example, runs a hoard walk every ten minutes. The volume daemon checks each volume to effect state changes every five seconds. Our code is structured such that volume version stamps are likely to be obtained or validated in the background by one of these daemons. This greatly reduces the chance that the cost of these tasks is incurred on demand during a user request.

4.3.1 Policy

As mentioned in Section 3, Venus should have some policy to determine when to obtain a volume callback. The optimal policy would obtain a volume callback only if a failure was going to occur and be repaired before the next remote update. Otherwise, either the volume callback would be broken, or the next validation would fail.

One could invent a variety of policies to approximate the optimal one. We decided to use a simple policy, in which Venus obtains volume callbacks only during hoard walks. We chose this policy for several reasons.

1. Volume version stamps are intended to be useful in preparing for failures. This is synonymous with the purpose of hoarding.
2. During a hoard walk, cached files are validated anyway. The additional overhead of obtaining a version stamp for each volume is low.
3. This strategy satisfies our scalability concerns. If a volume callback is broken, the client will not request another one until the next hoard walk.
4. Since hoard walks are periodic, the window of vulnerability to failures is bounded. For a client to lose the opportunity to validate files by volume, a remote update would have to be followed by a failure within one hoard walk interval (typically ten minutes). In this case, the client is no worse off than it was before the use of volume callbacks.

This policy also copes nicely with *voluntary* disconnections, when a user deliberately removes a laptop computer from the network. In our environment, many users have both desktop and laptop computers. While at work, they work from the desktop computers, leaving their laptops connected nearby. Some users modify files hoarded on their laptops from their desktop. Before disconnecting, they run a hoard walk on the laptop to fetch the files they just changed from the desktop. While connected, the laptop observes the remote updates to volumes that are referenced in its hoard database. These volumes are prime candidates for volume callbacks. A policy that becomes more conservative about obtaining volume callbacks when remote updates occur would be unlikely to obtain them in this case. In contrast, our policy takes advantage of explicit hoard walks as hints of imminent disconnection.

4.3.2 Access Rights

Directories in Coda have access lists associated with them that specify the operations that a user or group of users may perform on them. Venus caches access information to perform access checking locally. It obtains the information from the Vice status block, which is a result of most Vice calls. The access cache for a directory consists of a fixed number of entries containing a user identifier and that user's rights on the directory. Entries are considered valid when they are installed from the Vice status block. They are considered invalid (or suspect) if the object is invalidated, the user's authentication tokens expire, or if the AVSG grows.

When files are validated in groups, such as by volume, access information is not returned for the individual files. To avoid sending messages to the server to check access information, Venus must use the access cache more aggressively than it did in the past. If an object is deemed valid, clearly its access rights have not changed. Venus now considers entries in the rights cache for a file valid if the file is valid, and the entry corresponds to a user who is authenticated.

4.3.3 Effects of Replication

Coda's support of replicated volumes affects the client's handling of volume version state in two ways. First, Venus communicates with the AVSG as a group, sending the same copy of each request to each member of the group. This is performed by the underlying RPC protocol, which was designed to support remote procedure call to a set of machines in parallel. Because of this, a validation request must contain the stamps for all the servers in the VSG. Each server simply checks the one corresponding to it.

Second, Venus must collate multiple responses to its requests. When requesting version stamps, it must store the stamp for each server that responds. When validating version stamps, all servers must agree that the stamps are valid before Venus can declare them valid. Similarly, all servers must agree that a callback has been established before Venus can assume it has a callback on the volume.

5 Status and Evaluation

Servers supporting volume callbacks have been in use for several months. The corresponding Venus is currently in alpha test, and we expect to release it for production use shortly.

The primary reason for using large granularity cache coherence is to validate a client's cache quickly after a failure is repaired. In this section, we present

measurements of cache validation times for five typical Coda users under a variety of conditions.

5.1 Experiment Design

The time required to validate a client's cache after a failure is the figure of merit for our experiments. We call this the *recovery time* of the cache. Obviously, recovery time depends on the contents of the cache. For the experiments, we gathered the *hoard profiles* of five Coda users, summarized in Table 1. A hoard profile is the input to a program that updates the HDB. These profiles are used primarily for laptops. To broaden our study, we deliberately chose users whose profiles were dissimilar.

We performed the experiments with a single client and server, both DECstation 5000/200s with 32 MB of memory, running Mach 2.6. The client used a 50 MB Coda file cache. The machines were connected via Ethernet. To emulate slower networks and inject failures, we used a *failure library* linked into Venus and the server. The library allows packets to be delayed or suppressed according to a *filter*, which specifies under what conditions the mischief is to occur. For example, one might request packets to a certain host be dropped with some probability, or delayed as if the network were a lower speed. Requests to insert and remove filters are issued to the failure package via RPC.

We began each experiment by initializing the hoard database with the profiles for a single user. Then we ran a hoard walk, and partitioned the client from the server. Once the client detected the failure, we healed the partition, caused the client to notice the server was up, and immediately ran a hoard walk. We measured the time it took for Venus to validate its cache entries, from when it noticed the server was up to the end of the hoard walk. We assume no updates on cached volumes were made to the server by any other client during the failure. Although this is the best case, we believe it is an important common case in intermittent environments.

5.2 Parameters Explored

We studied recovery times over four network speeds and three validation strategies for each user. The network speeds were 10 Mb/sec, representing Ethernet; 2 Mb/sec, representing packet radio (such as NCR WaveLanTM); 64 Kb/sec, representing ISDN, and 9.6 Kb/sec, representing a typical dialup connection. The validation strategies were "NoOpt", "Batched", and "VCB". The NoOpt strategy validates an object by fetching its status block from the server and comparing it to the cached copy. This corresponds to the Vnode

Volume Type	Number of Files Cached				
	User 1	User 2	User 3	User 4	User 5
X11	38	127	133	125	142
T _E X			560	158	
System	9	6	190	342	689
Cboard					361
Other tools		42	13	13	42
Coda binaries			2	6	4
Coda sources			4	549	6
Kernel sources					24
User 1 personal	114				
User 2 personal		234			
User 3 personal			190		
User 4 personal				220	6
User 5 personal					537
Other personal	107	4	5	10	10
Total files	268	413	1097	1423	1821
Total volumes	7	6	9	11	12
Cache size (MB)	2.4	16.5	9.2	37.3	22.3

Table 1: Contents of Hoard Profiles for Five Coda Users, by Volume

This table characterizes the contents of the hoard profiles for the five Coda users studied in the experiments described in Section 5.1. Entries represent the number of files hoarded from each volume by each user.

The system volume contains system binaries, utilities, and include files. Cboard is a project volume for a calendar program; its maintainer is user 5. “Other tools” refers to five volumes containing utilities such as GNU-Emacs and less. The “Coda binaries” volume contains Coda-related programs that many users hoard. The “Coda sources” category is of interest primarily to Coda developers. It consists of two volumes containing scaffolding for the project tree, libraries, include files, and sources. User 4’s personal files are split into a home volume and a volume solely for object files. “Other personal” is a set of five volumes belonging to users other than the ones we studied. Two of those volumes contain versions of kermit that most users hoard, and one contains a popular window manager.

operation `GetAttr` [6]. The Batched strategy allows a group of files to be validated in one RPC. More specifically, in Coda up to 50 fids may be piggybacked with version information on a `GetAttr` request. The VCB strategy validates objects by volume using previously cached version stamps. These are also batched; for these experiments only 1 RPC is needed to validate the volumes.

Although the current production version of Coda uses the Batched strategy, we measured the NoOpt strategy for two reasons. First, it allows our results to be compared to file systems that do not batch validations, such as AFS. Second, even though batching takes less time and bandwidth at any speed than NoOpt, it has some disadvantages at low bandwidth.

Batching can result in large request packets – nearly 3KB in Coda. These requests stress the underlying RPC protocol, because retransmissions at low bandwidth can starve other requests, and cause Venus to declare servers down. Indeed, we experienced such failures while conducting our experiments! It may be more appropriate to use a smaller batching factor for low bandwidth networks. Latency is also significantly affected by request size when bandwidth is low. Currently a demand (user) request for one file will cause a batch validation of up to 50 files, which incurs additional latency that could be deferred to background processes.

Batching of volume validations does not have as great an impact on the system as batching of file

Network Speed	Validation Strategy	Recovery Time in Seconds					Relative Times
		User 1	User 2	User 3	User 4	User 5	
10Mb/s	NoOpt	6.8 (.5)	9.9 (.8)	20.9 (.6)	31.5 (.5)	46.0 (1.1)	100.0%
	Batched	2.5 (.5)	3.6 (.5)	8.2 (.5)	11.0 (.0)	19.0 (.8)	38.5%
	VCB	2.5 (.5)	3.5 (.5)	7.4 (.5)	10.0 (1.3)	17.5 (.8)	35.5%
2Mb/s	NoOpt	6.5 (.5)	11.0 (2.6)	21.3 (1.2)	32.0 (.5)	46.0 (.9)	100.0%
	Batched	3.0 (.0)	4.1 (.4)	9.4 (.5)	12.6 (.5)	21.0 (.8)	42.9%
	VCB	2.3 (.5)	3.7 (.5)	7.3 (.5)	9.4 (.5)	18.1 (.8)	34.9%
64Kb/s	NoOpt	12.8 (1.4)	17.5 (.5)	40.9 (1.4)	63.6 (1.6)	87.5 (2.2)	100.0%
	Batched	5.4 (.5)	7.2 (.5)	16.9 (.4)	24.3 (.5)	36.5 (.9)	40.6%
	VCB	2.3 (.5)	4.0 (.5)	7.4 (.5)	9.6 (.5)	17.8 (.9)	18.5%
9.6Kb/s	NoOpt	67.8 (1.4)	102.8 (.9)	226.1 (2.2)	342.4 (4.0)	453.8 (9.7)	100.0%
	Batched	23.8 (2.8)	31.4 (2.5)	80.9 (15.8)	103.1 (9.7)	136.3 (8.7)	31.5%
	VCB	4.8 (.5)	5.3 (.5)	8.9 (.6)	11.3 (.5)	20.3 (.9)	4.2%

Table 2: Cache Recovery Time (Seconds)

This table presents the time in seconds needed by a client to validate cached files when it discovers a server is up. The cache contents are determined by the hoard profiles for each of the five users. The rightmost column is the average reduction in validation time compared to NoOpt for each of the other two strategies. The reduction is given as a percentage, and is calculated as $(100 \times t_{\text{Other}})/t_{\text{NoOpt}}$. These results are conservative in a number of respects, as explained in Section 5.5.

The experiments were conducted with DECstation 5000/200s as the client and server, and volumes stored at one server. Measurements were taken over an Ethernet; for the three slower speeds, an emulator was used to delay packets. Each entry is the mean and standard deviation (in parentheses) of the most consistent eight trials from a set of ten.

validations because clients have information on many fewer volumes than files, and volume identifiers and version stamps are much smaller than their counterparts for files.

5.3 Results

Our results confirm that VCB compensates successfully for the reduction in bandwidth. Table 2 presents our observations. For all users and networks, recovery times are smallest using VCB, followed by the Batched and NoOpt strategies. There is variation across users proportional to the number of files cached. The improvement increases as bandwidth decreases. At 9.6 Kb/sec, where VCB is likely to be most important, recovery time takes only 4–7% of the time required by NoOpt, and 11–20% of the time required by batching. At higher bandwidths, the value of VCB diminishes, but it is always at least as good as the other two strategies. A glance at Table 2 reveals that the results for VCB at 9.6 Kb/sec and 10Mb/sec are not significantly

different.

An unexpected result was that the recovery time using VCB on a slow network was not constant over all users. We expected the bottleneck in this case would be the network. Since only one RPC was required to validate the volumes, we thought the recovery times would be similar. We observed recovery times proportional to the number of files cached, indicating the bottleneck is Venus. Most of its time is spent on two tasks: marking cached objects suspect when the server appears up, and performing the hoard walk, which involves iterating through all of the objects in the cache to ensure they are valid.

The number of callbacks at the server can be derived from Table 1, from the number of objects and volumes each user hoards. In these experiments, clients using the Batched or NoOpt strategies obtain callbacks for each file validated. Clients using VCB obtain callbacks only for the volumes they validated. The number

of callbacks obtained by clients using VCB is less than 3% of the number obtained by the other two strategies.

The results presented are for the case in which all validations succeed. Over longer periods, or for more active volumes, some validations will fail because of updates at the server. As long as some validations succeed, VCB will still perform better than the other strategies. The only case for which VCB is worse is if every volume validation fails, and then it is worse by 1 RPC. Considering what users hoard, this case is unlikely.

The volumes most likely to change are the personal or project volumes of other users, as shown in Figure 2. All of the users we studied hoard files from other user volumes; however, in all but one case they represent less than 1% of the total files. Therefore validating these files individually if necessary does not have a large impact on recovery time. Further, some user volumes were inactive during the period shown in Figure 2.

The next most frequently changed set of volumes are the Coda and kernel source volumes, which are shared by up to six project members. These change relatively slowly; Figure 2 indicates that the most active of these volumes, the Coda source area, was completely unchanged for half of the days in the period we studied. Since update traffic is bursty, the results from Figure 2 are conservative, especially for intermittent environments. Thus we are confident that the benefits listed in Table 2 are realistic.

5.4 Overhead

Of course, fast validation isn't free. There are several sources of network overhead caused by volume callbacks – obtaining callbacks, breaking callbacks, and validating volumes. Obtaining a callback on a volume requires validation of every cached file in the volume. Since this is already done by hoard walks, and the number of volumes is small compared to the number of files cached by clients, the additional overhead to obtain the volume callback is low.

In the worst case, all the volumes from which a client has cached files are being updated actively. The client then loses every volume callback it obtains, and its volume validations fail. If the sharing is false, the effort expended to get volume callbacks is wasted. Fortunately, callback requests and breaks are small messages, well under 100 bytes. Since these occur only once in every hoard walk period, the network overhead is still low. The failed volume validation costs one extra RPC. For volumes from which many files are cached, the cost of validating the files renders that RPC

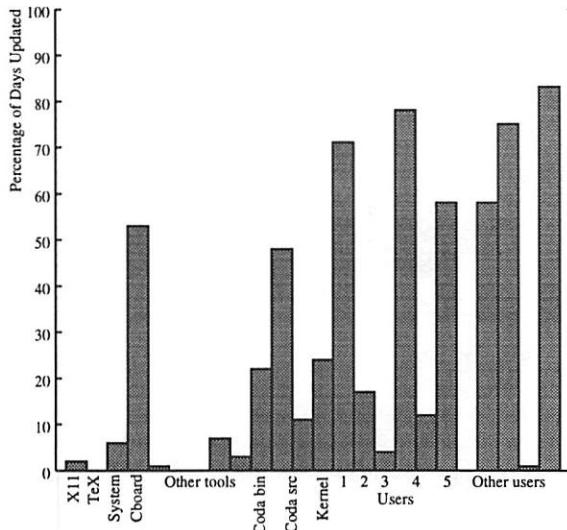


Figure 2: Daily Update Frequency of Volumes

This figure shows how often volumes used in our experiments are updated on a daily basis. The data was gathered from daily backup logs from January through March 1994. Each bar indicates the percentage of the days in the period during which at least one object in the volume was updated. We show volumes in the "Other users", "Other tools", and "Coda sources" categories separately, as well as both of User 4's personal volumes.

insignificant. If the sharing is real, the overhead due to volume callbacks is likely to be insignificant compared to the cost of re-fetching the shared data. Overall, the benefits of volume callbacks far outweigh the costs.

5.5 Accuracy of Results

The results in Table 2 underestimate the benefits of VCB in a number of respects. First, our failure library underestimates the delay for a given network speed. Emulation is performed by a package which intercepts outgoing packets and delays them based on the size of the packet, the network speed requested in the filter, and the delays for any packets queued ahead of the one to be sent. The delay is a simple-minded calculation, and does not take into account overheads such as UDP and IP header sizes, or IP fragmentation. A comparison of emulated and real times at 9.6Kbps is shown in Table 3.

Second, we used volumes with only one replica, when most volumes in Coda are triply replicated. Since many networks do not support multicast, an RPC request to an AVSG with more than one member is currently sent as separate messages to each member. If the network is the bottleneck, the time required to validate

Packet Size	Time (seconds)	
	Emulated	Real
60	.11 (.01)	.33 (.01)
260	.43 (.03)	.76 (.04)
560	.96 (.01)	1.4 (.0)
1060	1.8 (.0)	3.3 (2.6)
2060	3.5 (.0)	4.6 (.28)
3060	5.2 (.0)	6.6 (.0)
4060	7.9 (1.9)	8.7 (.0)

Table 3: Emulated vs. Real RPC at 9.6 Kbps

This table compares the round trip time for an RPC request and response of the same size, using the network emulator set to 9.6 Kbps over an Ethernet, and using a dialup SLIP link nominally rated at 9.6 Kbps. The experiments were conducted using an i386-based laptop as the client and a DECstation 5000/200 as the server. RPC packet headers are 60 bytes long; the first line gives the times for a null RPC. We show the mean and standard deviation for the most consistent eight trials from a set of ten. The large standard deviations for 4060 bytes (emulated) and 1060 bytes (real) were due to retransmissions during one or more runs.

cached files for each of the strategies in Table 2 will be proportionately larger.

Last, caches typically contain more than what is hoarded. This occurs for several reasons – name space exploration, objects left over from other tasks, and execution of a task to find files not included by hoard profiles.

Each of these effects underestimates the savings due to VCB, especially over low bandwidth networks.

6 Conclusion

This work was motivated by the demands of mobile computing. Large granularity cache coherence is valuable in that context because it allows a high level of consistency to be preserved even when communication is intermittent or expensive. But we anticipate that this mechanism will have broader applicability. For example, we expect it to be valuable in systems such as AFS, where recent measurements indicate over 50% of requests to servers are for fetching status [12]. We conjecture that a significant fraction of these are validation requests for files that once had callbacks. These callbacks may have been lost due to failures or expiry, since AFS-3 callbacks are effectively *leases* [3].

Another argument for maintaining cache coherence at a large granularity has been put forth independently by Wang and Anderson [13]. They proposed

maintaining cache coherence on clusters of files, such as subtrees. Their primary motivation is to reduce server state rather than communication.

Regardless of specific motivation, we are convinced that large granularity cache coherence is a practical and important technique for distributed computing. Our experience and measurements confirm that it is valuable in preserving the quality of file access in intermittent networking environments. Large granularity cache coherence costs little, and offers the potential for big savings.

Acknowledgements

We wish to thank members of the Coda group, in particular Maria Ebling, Puneet Kumar, Brian Noble, and David Steere, and the members of our user community, especially David Eckhardt. We also wish to thank our referees, our shepherd Mike Jones, Randy Dean, Hugo Patterson, and Mirjana Spasojevic for their helpful comments.

References

- [1] R. Alonso, D. Barbara, and L. Cova. Using Stashing to Increase Node Autonomy in Distributed File Systems. In *Proceedings of the Ninth Symposium on Reliable Distributed Systems*, October 1990.
- [2] Rick Floyd. Short-Term File Reference Patterns in a UNIX Environment. Technical Report TR 177, Department of Computer Science, University of Rochester, March 1986.
- [3] Cary G. Gray and David R. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *The Twelfth ACM Symposium on Operating Systems Principles*, pages 202–210. ACM, December 1989.
- [4] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.
- [5] James J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1), February 1992.
- [6] S. R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *USENIX Summer Conference Proceedings*. USENIX Association, 1986.

- [7] Puneet Kumar and M. Satyanarayanan. Log-Based Directory Resolution in the Coda File System. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pages 202 – 213, January 1993. Also available as technical report CMU-CS-91-164, School of Computer Science, Carnegie Mellon University.
- [8] L. Mummert and M. Satyanarayanan. Variable Granularity Cache Coherence. *Operating Systems Review*, 28(1), January 1994.
- [9] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Knupfer, and James G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, December 1985.
- [10] M. Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39(4), April 1990.
- [11] R.N. Sidebotham. Volumes: The Andrew File System Data Structuring Primitive. In *European Unix User Group Conference Proceedings*, August 1986. Also available as Tech. Rep. CMU-ITC-053, Carnegie Mellon University, Information Technology Center.
- [12] Mirjana Spasojevic and M. Satyanarayanan. A Usage Profile and Evaluation of a Wide-Area Distributed File System. In *USENIX Winter Conference Proceedings*. USENIX Association, January 1994.
- [13] Randolph Y. Wang and Thomas E. Anderson. xFS: A Wide Area Mass Storage File System. In *Proceedings of the Fourth Workshop on Workstation Operation Systems*, pages 71 – 78, October 1993.

Author Information

Lily Mummert received the B.S. and M.S. degrees in computer science from Texas A&M University in 1985 and 1986, respectively. She is currently a Ph.D. student at Carnegie Mellon University, working on using weak connectivity in distributed file systems. Prior to joining the Coda project, she worked on the Camelot distributed transaction facility, and on file reference tracing.

Mahadev Satyanarayanan is an Associate Professor of Computer Science at Carnegie Mellon University. He is currently investigating the connectivity and resource constraints of mobile computing in the context of the Coda File System. Prior to his work on Coda, he was a principal architect and implementor of the Andrew File System. Satyanarayanan received the PhD in Computer Science from Carnegie Mellon University in 1983, after a Bachelor's degree in Electrical Engineering and a Master's degree in Computer Science from the Indian Institute of Technology, Madras. He is a member of the ACM, IEEE, Sigma Xi, and Usenix, and has been a consultant to industry and government.

An Analysis of Trace Data for Predictive File Caching in Mobile Computing

*Geoffrey H. Kuenning, Gerald J. Popek, Peter L. Reiher
University of California, Los Angeles**

Abstract

One way to provide mobile computers with access to the resources of a network, even in the absence of communication, is to predict which information will be used during disconnection and cache the appropriate data while still connected. To determine the feasibility of this approach, traces of file-access activity for three diverse application domains were collected for periods of over two months. Analysis of these traces using traditional and new measures reveals that user working sets tend to be small compared to modern disk sizes, that users tend to reference the same files for several days or even weeks at a time, and that different users do not tend to write to the same file except in highly constrained circumstances. These factors encourage the conclusion that an automated caching system can be built for a wide variety of environments.

1 Motivation

The value of mobile computers is that they allow users to work while disconnected from their normal resources. However, mobile computers typically have a great deal less disk storage than is available via remote mounting on connected networks. This forces mobile computer users to face a challenging problem of making sure their limited disks always store the information they will need while disconnected from other machines. Requiring users to deal explicitly with this issue puts a heavy burden on them, and the realities of modern software methods make it nearly impossible for users to identify all the files they actually need.¹ A fully automated caching mechanism that predictively stored all files a user needs on his mobile machine

*This work was partially supported by the Advanced Research Projects Agency under contract N00174-91-C-0107.

¹For example, starting the X Window System requires access to 10–30 files or more. The identities of many of these are surprising even to expert systems programmers [6].

would be very valuable. Such a mechanism is only practical, however, if information that can be gathered automatically fully captures the typical user's working set of files.

A prototype system of this sort was developed under CMU's Coda system [6, 14] and proved successful, but was inconvenient for the user and was tested only in one application environment.

We undertook this research to investigate the practicality of automatic file caching for mobility in a wider set of application domains, and to discover new and less-burdensome ways of identifying files to be cached. Our approach was to collect traces of file-access activity in several environments over a long period of time, and analyze them for feasibility and predictability of caching.

We chose to collect our own traces, rather than using existing traces, for three reasons. First, few existing traces are long enough. Because most existing traces collect read/write activity, a few weeks of data is sufficient to tax resource limits. We were interested in observing longer-term periodic behaviors such as end-of-the-month billing work in an accounting department, which therefore required a several-month trace to establish a pattern.

Second, existing traces have tended to be limited to an engineering application domain, usually programming. We wanted to investigate the behavior of non-programmers as well, in the twin beliefs that this type of user will eventually be the largest population of portable users, and that these users may behave quite differently from programmers.

Third, most previous studies have generally been limited to analysis of working-set sizes and file-system performance data [1, 2, 6, 11, 14]. The latter is not relevant to this research, and the former, while very important, is not in itself sufficient to characterize the user behaviors critical to successful mobile caching.

Successful automated caching requires two charac-

teristics in user behavior:

- The working set of files, as observed over a period of days or weeks, must be small enough to fit on a portable's disk.
- It must be possible to predict the working set in advance, using hints such as the current working set, historical file access patterns [15], or known patterns in user behavior.

Analysis of the data we have collected shows that these characteristics are present in a number of different application domains.

2 Methodology

We collected our traces at Locus Computing Corporation, a software development and consulting firm, during the summer of 1993. One of Locus' products, PC/Interface (PCI) [8], is a DOS-to-UNIX file system implemented as a pseudo-disk driver on a DOS machine which communicates via Ethernet to a file server on the UNIX system, making the UNIX file system available to the DOS users as native PC files. In the environments monitored, the local DOS filesystem was used to store some applications software, but all shared corporate data was accessed via PCI. The UNIX server for PCI was modified to log opens, closes, and deletes of files. By avoiding read/write logging, we minimized the performance impact and kept the log files small. Log entries contain an operation type and subtype (e.g., open for read), the UNIX timestamp in seconds, the UNIX UID of the invoker, the process ID, the absolute pathname of the file, and the size of the file.

Three different user environments were monitored. In the first, referred to as "personal productivity," the server was a machine that acted as the network filesystem for 47 users running business-oriented applications such as e-mail, project and calendar scheduling, and word processing. These users did not tend to store important files on their own machines, so they generated high activity at the server. This server was traced for 1563 hours (65.1 days, or 9.3 weeks),² recording 4,637,924 accesses.

In the second environment, referred to as "programming," the server was a cluster of 10 machines running IBM's Transparent Computing Facility, an adaption of the Locus distributed operating system [12], which provides a single-system image to users of multiple

²50 days into this trace, there was a data gap of approximately 48 hours due to an administrative error. It does not appear that this gap affects the validity of the analysis.

machines. Each machine ran a separate PCI server, and logs from these servers were later combined for analysis. Most of the users of this server were programmers working on DOS-based software. Because they performed much of their work locally, accessing the shared server mostly to retrieve or update shared source files, they generated relatively little server activity. The traces on this server essentially reflect commits to a shared database, while omitting most localized file activity. This server was accessed by 64 users and was traced for 1693 hours (70.5 days, or 10.1 weeks), recording 93,719 accesses.

In the third environment, referred to as "commercial," the server was a single machine used by the accounting department to run a commercial accounting application. The master corporate accounting database was kept on the UNIX server, but all access to this (shared) database was via DOS workstations running the commercial package. This server was accessed by 7 users and was traced for 1257 hours (52.4 days, or 7.5 weeks), recording 371,830 accesses.

The nature of the traced environment (local files stored on PC's, with shared files stored remotely) parallels the expected behavior of mobile users, who will probably store heavily-used applications locally³ but make extensive use of shared resources when they are network-connected. However, based on preliminary analysis of these traces, we also generated two modified traces that omitted certain characteristics we felt might be absent on portable platforms due to different software and user behaviors. For the commercial environment, we reduced all file sizes to a maximum of 1 MB, on the theory that very large databases would be represented by smaller slices in a portable environment. This change primarily affected the statistics on working-set sizes and the amount of data involved in *write conflicts* and *attention shifts*, which are measures of file sharing and working-set variability that we will define in Section 3. For the productivity environment, we eliminated all references to fax spooling and mail files, because such files are handled in a queued (as opposed to shared) manner in disconnected environments. This change affected all of the statistics we analyzed. These two data sets are referred to as the "reduced commercial" and "reduced productivity" environments in the tables and graphs.

Once the traces were collected, we canonicalized them using a simple awk script that converts relative pathnames to absolute form, correlates each close with the corresponding open and produces an output line whose format is independent of the operation type to make subsequent processing easier. These canonical-

³We hope that even these will eventually fall under the purview of an automated caching system.

ized files were then compressed and used as the basis for our analysis. The largest of these files (from the productivity server) is nearly 18 megabytes in its compressed form, and about 10 times that large when expanded.

Originally, we used a collection of shell and awk scripts for all analysis. As the collected data grew, many of these scripts became computationally impractical and were replaced by tailored programs. The current design performs the analysis in two phases. First, a single-pass program reads the data and extracts summary information of interest. For example, for each 24-hour day in the collected data, the extraction program writes a single line for each user giving the total size of that user's working set, measured in both megabytes and files. A second pass then analyzes these summary files with general-purpose statistical tools, generating the final tables and graphs presented in this paper.

3 Statistics

We generated the same statistics for each parameter in each environment: mean, standard deviation, and maximum. Besides the traditional measure of working-set size, we looked at two measures that have special application to mobility: *write conflicts* and *attention shifts*.

We define a write conflict event to occur when two users write to the same file within a relatively short time span. In a mobile environment, a conflicted file might be replicated on two or more computers, and the system would be required to automatically resolve these conflicts after the fact in a manner similar to the Ficus distributed file system [3, 4, 7, 13], to force the user to resolve them by hand [6], or to limit writing to only one user. We examined conflicting writes within a 24-hour period (corresponding to taking a machine home overnight) and a 7-day period (corresponding to traveling with a machine).

An attention shift occurs when a single user radically changes his or her working set. We identified attention shifts by looking at the working sets in successive active n -hour time periods (which did not necessarily represent adjacent days or weeks). Within each time period, we counted the total numbers of files accessed, k_1 and k_2 , and then calculated $k = \min(k_1, k_2)$. Within the second period, we also counted the total number m of files that had not been referenced during the first period, but that had existed prior to either period.⁴ An attention shift was defined to occur if $m \geq pk$, where $0 \leq p \leq 1$. Attention shifts can be characterized by

⁴We eliminated files that were created during the second period because they are not problematical for a caching system that must predict which existing files need to be stored.

the parameters p , expressed as a percentage, and n , the number of hours in the period. We use the notation $p\%/n$ to describe an attention shift parameter pair. Based on a sensitivity analysis (see Figures 6–8), we chose $p = 20\%$. We chose $n = 24$ and $n = 168$ (1 week) because these represent typical disconnection periods for many portable users.

A final characteristic of an attention shift is the *age* of the shift, which represents the amount of time which has elapsed since the user last referenced one of the "new" files. We estimated the age by locating the most recently-referenced "new" file (a file included in count m), and subtracting its reference time from the start time of the second period. This is a conservative measure, since it assumes that the most-recently-referenced file is representative of the entire group m of "new" files.

However, since many of the newly-referenced files did not appear previously in the trace, it was not always possible to find a file to use in calculating the age of the shift. In this case, we conservatively assumed that the "new" files had been referenced exactly one second before the beginning of the entire trace. Because of these two assumptions, the attention-shift ages reported in this paper are only a lower bound on the true ages that would be encountered by a predictive caching system.

The *bounded locality intervals* discussed in [9] are similar to attention shifts, but are parameterized on working-set sizes rather than on the expected length of a disconnection.

The statistics we report are:

Working-set statistics. For each day and week, we calculated the working set size in files, MB, and number of accesses. Means and standard deviations were calculated by averaging data across time for each UID, and then calculating the mean and standard deviation across the per-UID means.

Attention-shift statistics. For each 1-day and 7-day attention shift, we examined the total size of the working set needed to hold both the old and the new data (in files and MB). We also calculated the per-user attention shift rate per day and per week. Finally, we calculated the age of each shift.

Conflict statistics. For each conflict, we examined the number of users involved and the size of the file involved. We also calculated the per-user conflict rate per day and per week.

Success in mobile computing depends on small values for all of these statistics. Clearly, the working set must be small enough to fit comfortably on the typical portable's disk. The attention-shift rate should remain

low, both so that the longer-period working set remains small and so that it is easier to predict the future working set based on recent behavior. The conflict rate must remain low to allow convenient file updates.

4 Analysis

The results of our analysis are very encouraging for our intended application, automated caching of files for mobile computers. As hoped, working sets are small and attention-shift rates are low. Conflict rates are generally low, and it is clear how one could handle conflicts in the environments that had high conflict rates. However, attention-shift ages tend to be high, indicating that a predictive caching system will need to exercise significant intelligence to ensure that a portable computer is prepared for attention shifts.

Each table of statistics given below lists the mean for the statistic, followed by the standard deviation (in parentheses) and the maximum. For example, in Table 1, the mean daily working set for the productivity environment was 1.0 MB, with a standard deviation of 2.0 MB and a maximum of 134.5 MB.

With the exception of Figures 6–8, all figures show the variation in a given measure over the duration of the trace. For example, Figure 1 shows the daily and weekly working sets for the productivity environment, for each day and each week captured during the trace.⁵

4.1 Working Sets

Table 1 summarizes the working-set sizes we observed. Figures 1–4 show the variation in mean and maximal working set sizes with time.

Mean working-set sizes tended to be small in all three environments, with the largest being about 18 MB per day and 24 MB per week, in the commercial environment. Maximal working sets were very large (148 MB per week) only in the personal-productivity environment, apparently due to a single grep-style operation that occurred in week 9. This “grep phenomenon” is clearly visible in Figure 1. Eliminating this single maximum produced a secondary maximum of only 76 MB. Maximal working sets in the other environments ranged only to 66 MB.

These working-set figures indicate that it will be easy to store enough files on a portable disk to satisfy the

⁵In these and all other graphs, the lines connecting data points are present only to make it easier to see associated points, and are not meaningful in themselves. In particular, although the daily maxima in the right-hand sides of Figures 4 and 5 appear to exceed the weekly maxima, careful examination shows that only the connecting lines cross, and the actual data points for weekly maxima are always larger than the daily values.

average user,⁶ although some software or user behavior may have to change. (For example, instead of relying on a large grep, a user might use an inverted index to locate the files containing references to a particular string [10].)

4.2 Attention Shifts

Tables 2 and 3 summarize the attention shifts observed. Figures 6–8 show the sensitivity of attention-shift rates to the parameter p . Except in the commercial environment, the number of attention shifts steadily decreases with increasing p , but the exact shape of the curve is quite inconsistent. In the absence of a clear-cut change in curvature (a knee or cliff), to guide us in the selection of p , we chose $p = 20\%$, which is near enough to the peak of the curves that we will not tend to underestimate the number of attention shifts, yet not so small that we will detect a shift every time a user accesses one or two new files.

Figures 9–11 show the variations in attention-shift rates with time, for $p = 20\%$. The amount of data involved in attention shifts was generally small (33 MB or less), though the maxima were large (up to 152 MB; this follows from the size of the maximal working set and the definition of an attention shift). In all three environments, the number of attention shifts was surprisingly large and consistent, averaging up to 0.6 per user per week. This has serious implications for a predictive caching scheme, because it shows that simply caching least-recently-used files is not sufficient.

However, because of the small size of the working sets involved in the average attention shift, a well-designed predictive cache can afford to store both the old and the new set, so that attention shifts need not affect the usability of a mobile computer.

Of course, if there is space to store both the old and new working set, the question arises whether a simple LRU scheme would be sufficient to ensure that both working sets are available. The attention-shift age figures shown in Tables 2 and 3 belie this notion. For both the programming and the reduced productivity environments, the mean age of an attention shift is over 4 weeks and the maximum is near the length of the trace, indicating that an LRU cache would very likely have been flushed by transient phenomena before the older files were re-referenced. This hypothesis is strengthened by the observation that the conservative method of estimating the ages of previously-unreferenced files,

⁶We expect working-set sizes to change dramatically over the next few years as users move towards multimedia applications, but we also expect that disk sizes will increase sufficiently for portable computers to keep pace. In some sense, this phenomenon is self-regulating, since users will not tend to use images and sounds extensively if this would tax their portable storage capacity.

Environment	Daily WS Size (MB)			Daily WS Size (Files)			Weekly WS Size (MB)			Weekly WS Size (Files)		
	Mean	σ	Max	Mean	σ	Max	Mean	σ	Max	Mean	σ	Max
Productivity	1.0	(2.0)	134.5	39	(80)	3293	2.7	(4.7)	148.4	110	(215)	3284
Reduced Productivity	0.7	(1.8)	41.1	7	(10)	547	1.4	(2.8)	43.6	19	(31)	548
Programming	0.3	(0.4)	18.0	10	(27)	2153	0.6	(1.1)	18.3	22	(55)	2170
Commercial	18.2	(13.1)	65.0	294	(442)	1643	26.8	(16.6)	65.7	374	(553)	1638
Reduced Commercial	10.9	(6.0)	33.6	294	(442)	1643	16.8	(8.7)	33.8	374	(553)	1638

Table 1: Working-Set Statistics

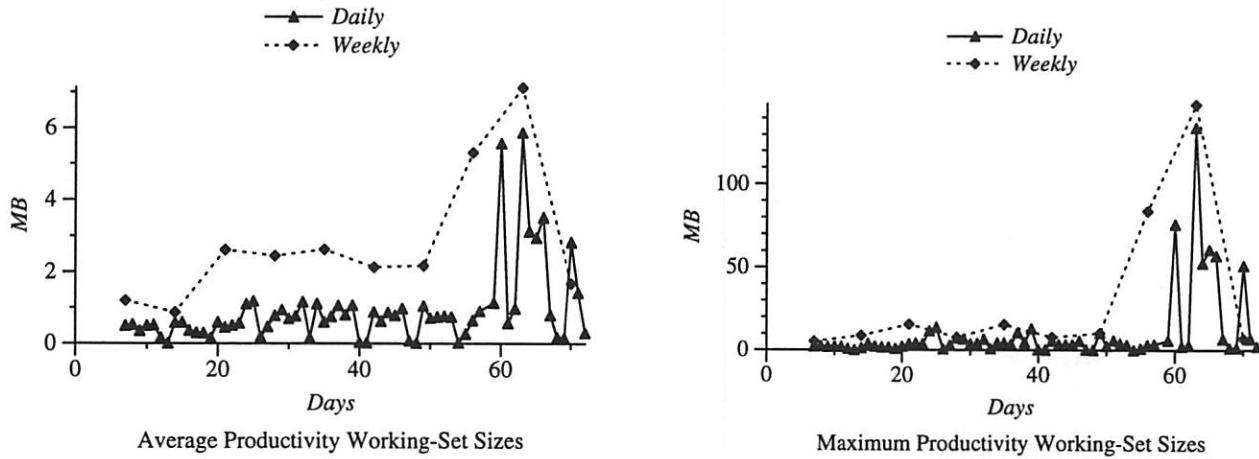


Figure 1: Working-Set Sizes for Productivity Environment

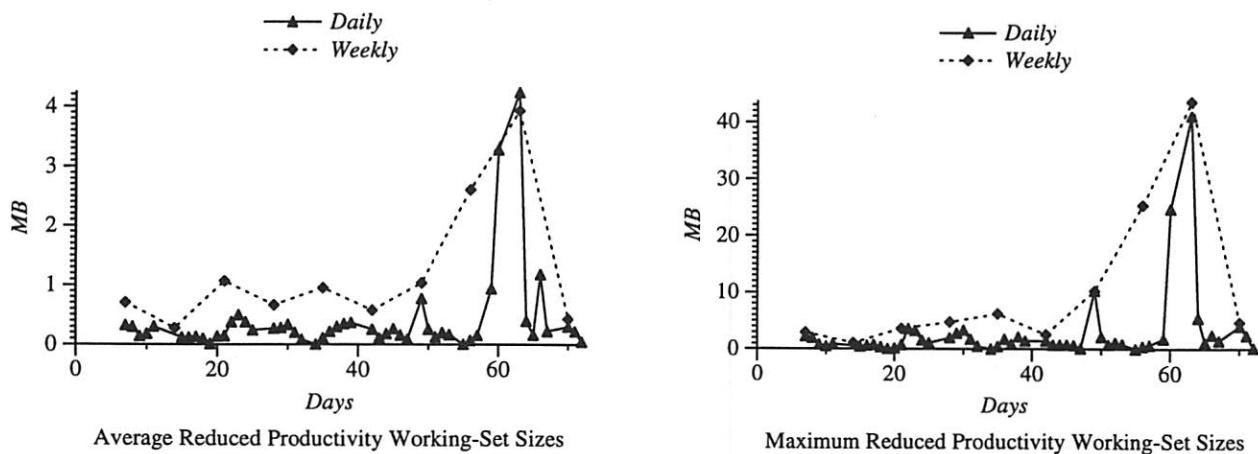


Figure 2: Working-Set Sizes for Reduced Productivity Environment

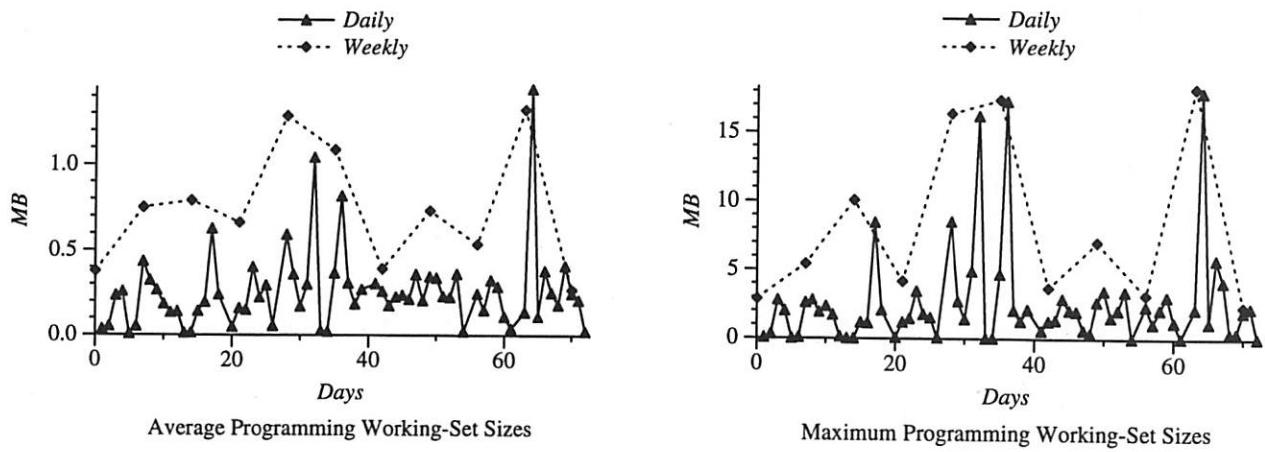


Figure 3: Working-Set Sizes for Programming Environment

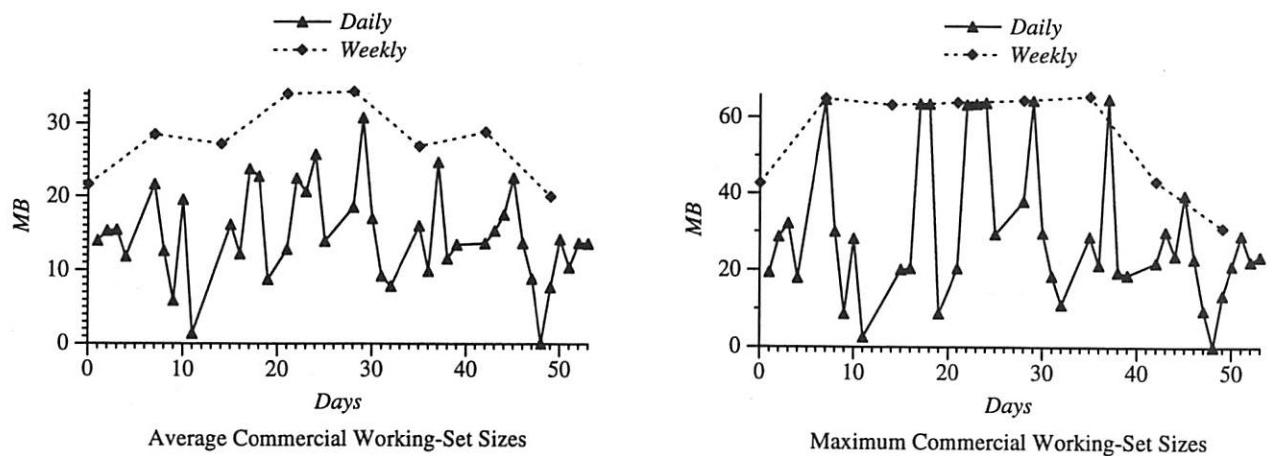


Figure 4: Working-Set Sizes for Commercial Environment

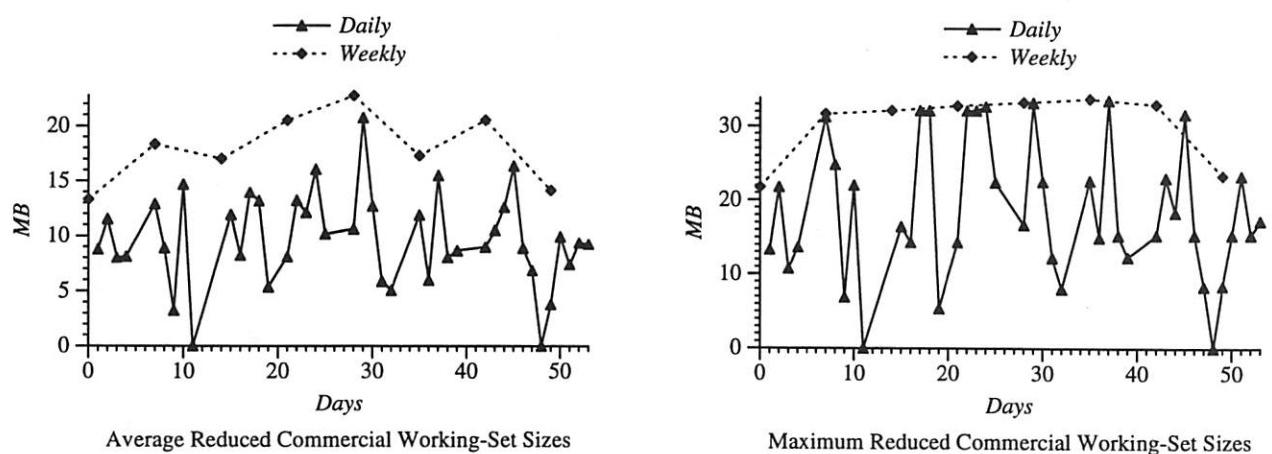


Figure 5: Working-Set Sizes for Reduced Commercial Environment

Environment	Number Per User Per Day			MB Involved			Files Involved			Age (Days)		
	Mean	σ	Max	Mean	σ	Max	Mean	σ	Max	Mean	σ	Max
Productivity	0.4	(0.3)	0.8	1.6	(6.5)	135.7	64	(164)	3296	10.0	(15.7)	64.7
Reduced Productivity	0.2	(0.2)	0.5	0.8	(3.2)	41.1	13	(33)	548	26.2	(19.7)	64.7
Programming	0.3	(0.2)	0.5	0.6	(1.6)	20.9	16	(109)	2161	28.0	(21.3)	70.2
Commercial	0.3	(0.3)	0.9	21.8	(13.8)	65.7	217	(398)	1654	3.2	(4.6)	35.7
Reduced Commercial	0.3	(0.3)	0.9	14.6	(8.1)	33.8	217	(398)	1654	3.2	(4.6)	35.7

Table 2: 20%/24-Hour Attention Shifts (All Users)

Environment	Number Per User Per Week			MB Involved			Files Involved			Age (Days)		
	Mean	σ	Max	Mean	σ	Max	Mean	σ	Max	Mean	σ	Max
Productivity	0.6	(0.3)	0.8	4.7	(12.4)	151.8	177	(376)	3423	15.7	(15.2)	62.7
Reduced Productivity	0.3	(0.2)	0.4	2.0	(5.5)	44.3	37	(71)	553	32.4	(18.9)	62.7
Programming	0.4	(0.2)	0.6	1.7	(3.4)	22.6	55	(215)	2174	28.9	(20.0)	68.2
Commercial	0.5	(0.4)	1.0	33.3	(17.4)	66.8	420	(584)	1661	11.1	(6.1)	33.7
Reduced Commercial	0.5	(0.4)	1.0	21.1	(9.0)	33.8	420	(584)	1661	11.1	(6.1)	33.7

Table 3: 20%/168-Hour Attention Shifts

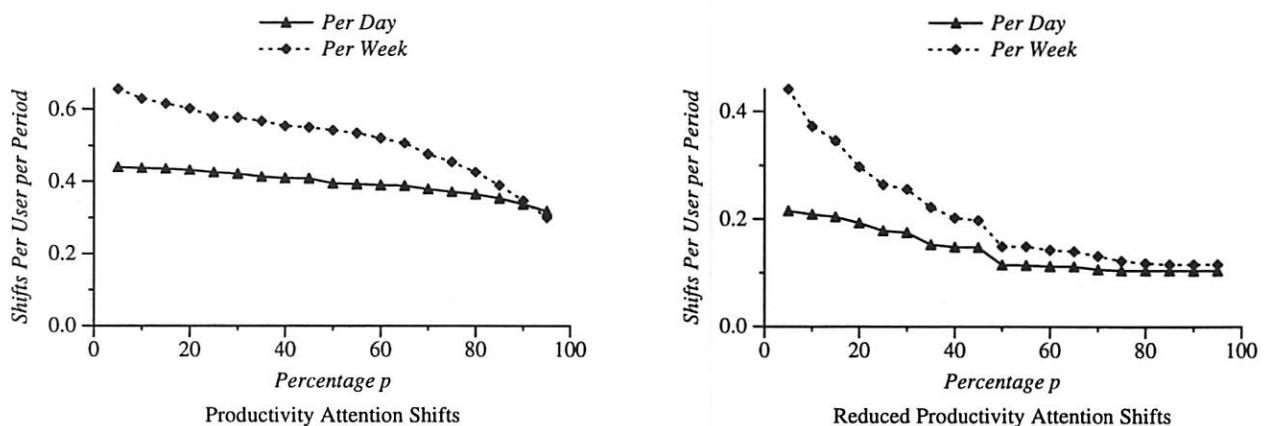


Figure 6: Attention-Shift Sensitivity for Productivity Environment

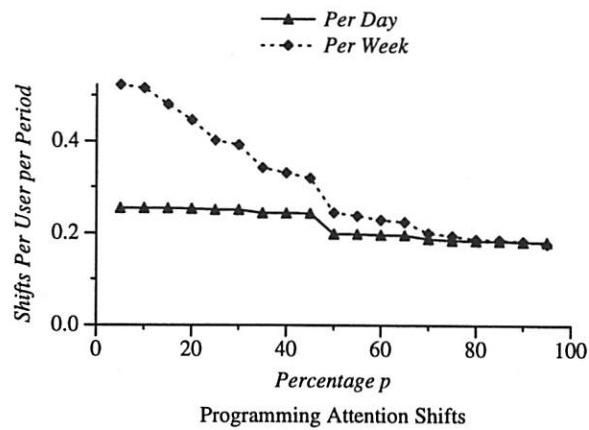


Figure 7: Attention-Shift Sensitivity for Programming Environment

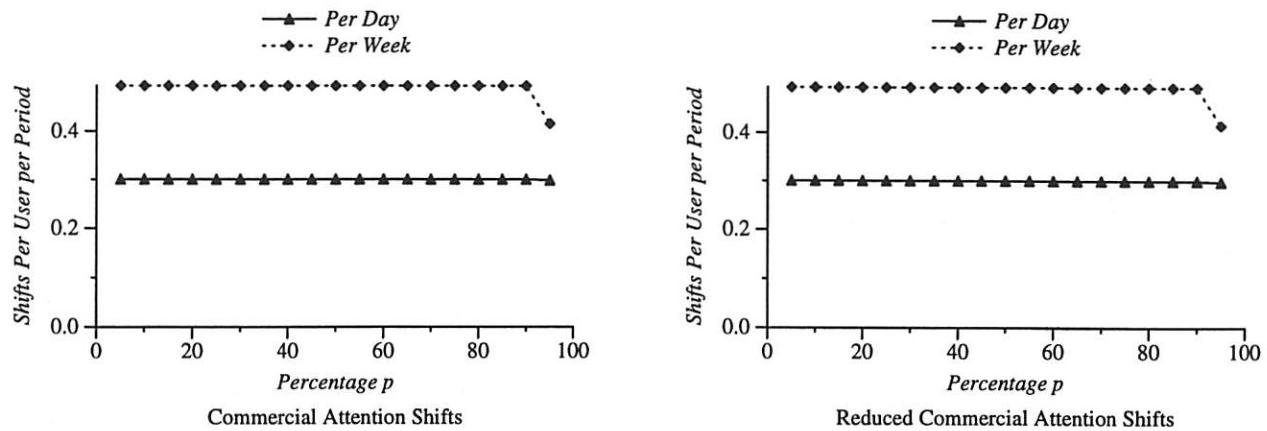


Figure 8: Attention-Shift Sensitivity for Commercial Environment

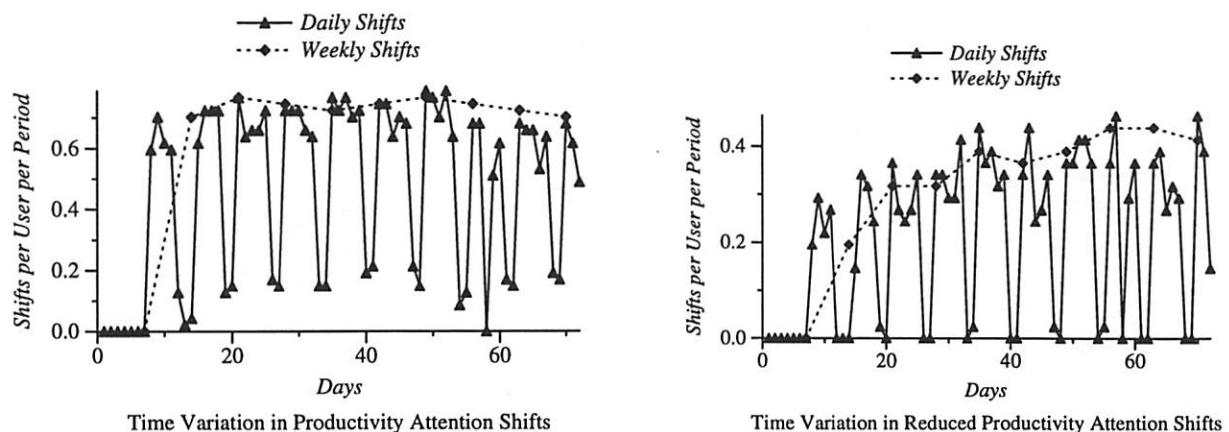


Figure 9: 20% Attention-Shift Rates for Productivity Environment

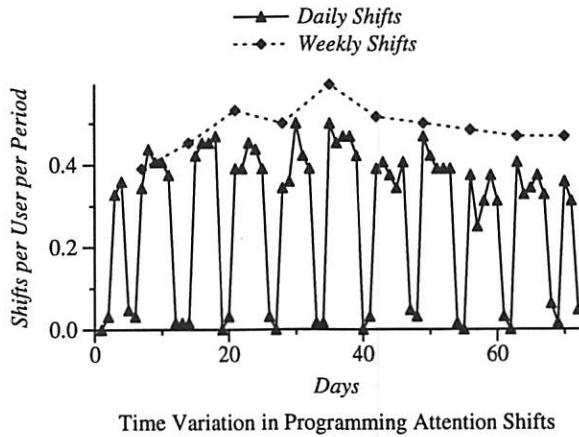


Figure 10: 20% Attention-Shift Rates for Programming Environment

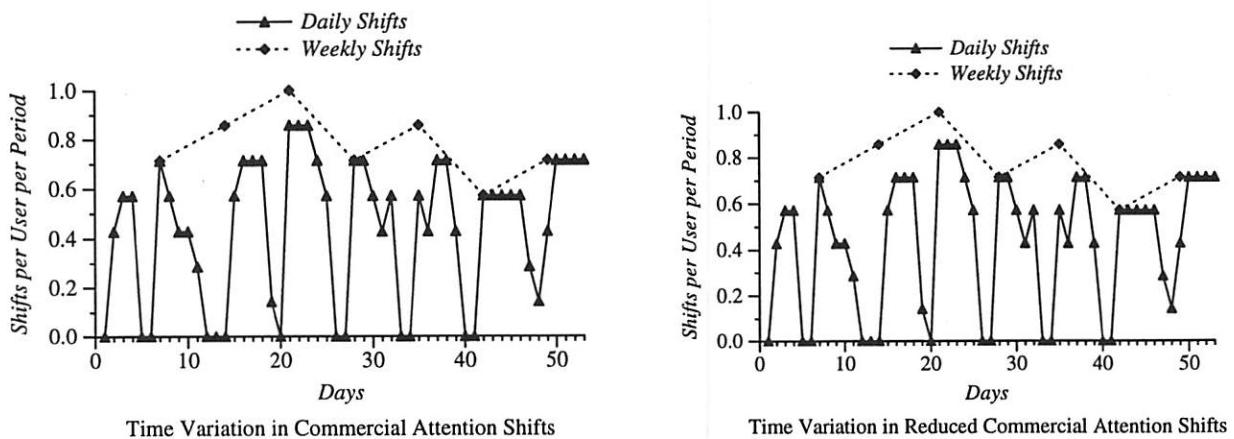


Figure 11: 20% Attention-Shift Rates for Commercial Environment

explained in section 3, would produce a mean age of approximately half the length of the trace (about 5 weeks) if there were absolutely no historical data in the trace. In actuality, the new working set may not have been accessed for many months and thus may have been flushed from even a very lengthy LRU cache. Other methods will be needed to ensure that a mobile machine will be prepared for an attention shift. The above data merely assures us that there will be room to store both today's and tomorrow's working sets once they have been identified.

4.3 Conflicts

Tables 4 and 5 show statistics about conflicts and their rate of occurrence, respectively. Figures 12–14 show the variations in conflict rates with time. Conflicts were very rare in the “programming” environment, averaging 0.01 conflict per user per day, and only 0.10 per week. In nearly every case only two users were involved in a given conflict, although occasionally a third would write to the same file within 24 hours.

As expected, the 7 users of the “commercial” environment, with its shared accounting database, produced a high conflict rate of 11 per user per week, with up to 6 users writing to the same file in a single day. In a mobile environment, an automated resolver similar to those discussed in [13] would be required to handle these numerous conflicts. Since accounting applications typically involve appending records to a transaction database, we expect that such a resolver would be easy to write.

The surprise was the “personal productivity” environment, which produced conflict rates up to 1.2 per user per day, with up to 22 users writing to the same file in a single 24-hour period. We examined these conflicts in more detail to discover the cause, and found that nearly all of them involved mailboxes or fax-spooling files.

Since both mailbox and spooling files operate in a modified append-only mode (all but one user appends to the end of the file, and a simple locking mechanism prevents update while other file contents are modified), this does not present a problem for mobility. In fact, the retry-on-failure queuing algorithm of mailers would handle mailbox conflicts with no software changes. In view of these observations, we generated the “reduced productivity” trace, which omitted these files from the statistics. With this change, the conflict rate dropped to only 0.04 per user per week, a number so small that it could conceivably be handled even without the help of automatic resolvers.

5 Future Work

Based on the above analysis, we expect to build a prototype caching system incorporating a prediction mechanism which, by observing user behavior, will calculate the current working set, detect attention shifts, and predict possible future working sets. A cache manager will then ensure that these working sets are available on the portable computer when it is disconnected from the network.

A cache miss during disconnection is a serious, often catastrophic event for a user who cannot continue to work in the absence of a critical file. There are only two real options for dealing with this case:

1. Provide enough alternate working sets that the user can shift to a secondary or tertiary task [6, 14].
2. Provide a foreground or background method that initiates communication (most likely expensive and slow) to retrieve the missing file [5].

We plan to provide both of these options in our prototype, though we hope to rely primarily on the first.

6 Conclusions

The data gathered and analysis performed in this study strongly indicate that predictive file caching for mobile computing is a feasible approach. However, the data also indicates that simple LRU caching is insufficient. Therefore, we conclude that more sophisticated automatic predictive file caching mechanisms will be required to make the file system of a mobile computer appear transparently the same as the file system of a desktop machine. We intend to investigate suitable algorithms for this purpose, guided by these results and by further analysis of our data.

Trademarks

PC/Interface is a trademark of Locus Computing Corporation. UNIX is a trademark of X/Open Company, Ltd.

References

- [1] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Sherriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pages 198–211. ACM, October 1991.

Environment	Daily Conflicts			Weekly Conflicts		
	Mean	σ	Max	Mean	σ	Max
Productivity	1.19	(1.16)	4.28	5.57	(3.58)	10.11
Reduced Productivity	0.00	(0.01)	0.05	0.02	(0.03)	0.07
Programming	0.01	(0.02)	0.06	0.10	(0.09)	0.28
Commercial	4.29	(4.74)	16.29	11.30	(8.92)	24.57
Reduced Commercial	4.29	(4.74)	16.29	11.30	(8.92)	24.57

Table 4: Conflict Rates

Environment	MB Involved in Daily Conflicts			Users Involved in Daily Conflicts			MB Involved in Weekly Conflicts			Users Involved in Weekly Conflicts		
	Mean	σ	Max	Mean	σ	Max	Mean	σ	Max	Mean	σ	Max
Productivity	0.02	(0.08)	2.05	3.39	(3.06)	22.00	0.02	(0.08)	2.05	3.61	(3.62)	27.00
Reduced Productivity	0.04	(0.04)	0.12	2.00	(0.00)	2.00	0.04	(0.04)	0.12	2.00	(0.00)	2.00
Programming	0.07	(0.16)	1.08	2.02	(0.15)	3.00	0.06	(0.12)	1.08	2.09	(0.29)	3.00
Commercial	0.22	(0.81)	5.37	3.16	(1.10)	6.00	0.27	(0.83)	5.37	3.16	(1.28)	6.00
Reduced Commercial	0.17	(0.81)	5.37	3.16	(1.10)	6.00	0.20	(0.83)	5.37	3.16	(1.28)	6.00

Table 5: Conflicts

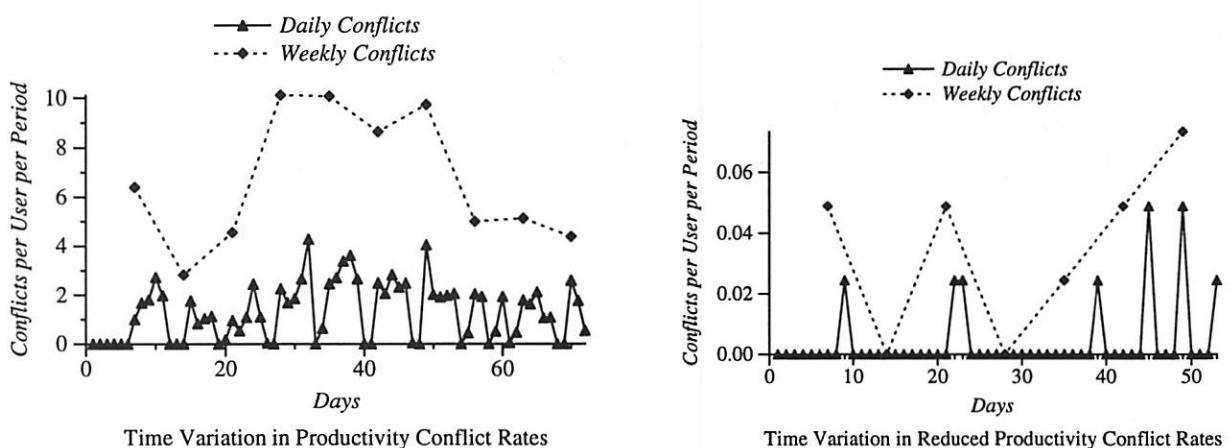


Figure 12: Conflict Rates for Productivity Environment

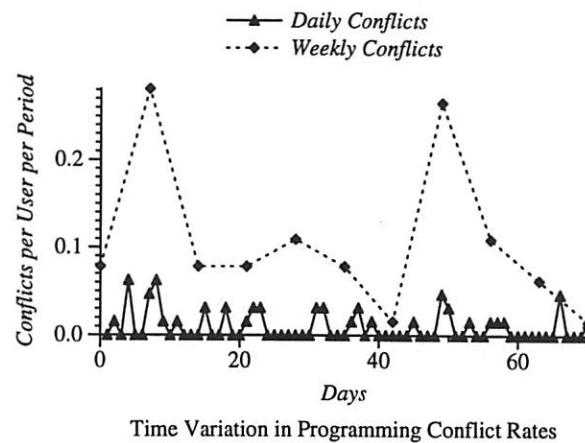


Figure 13: Conflict Rates for Programming Environment

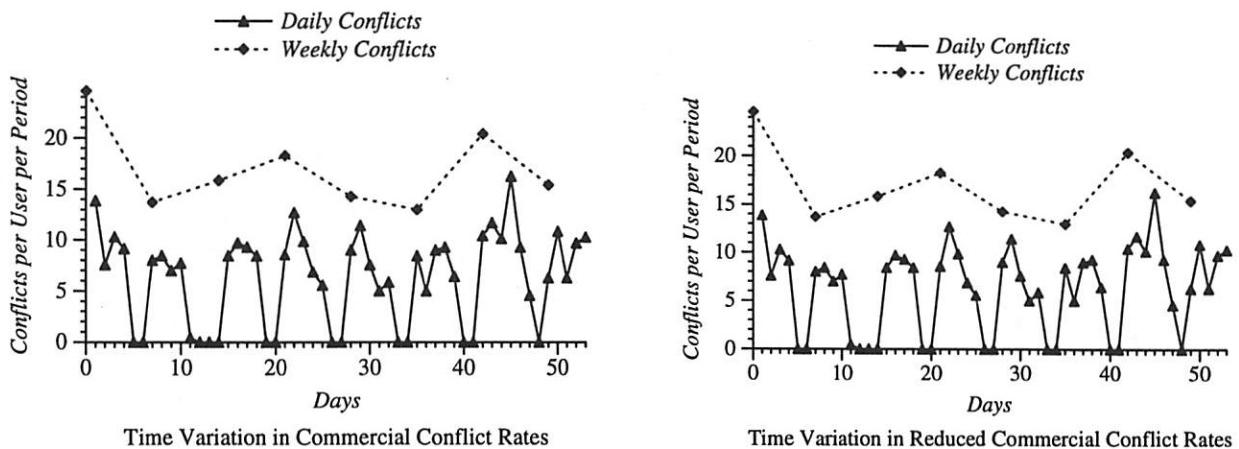


Figure 14: Conflict Rates for Commercial Environment

- [2] Matthew Blaze and Rafael Alonso. Dynamic hierarchical caching for large-scale distributed file systems. In *Proceedings of the Twelfth International Conference on Distributed Computing Systems*, pages 521–528, June 1992.
- [3] Richard G. Guy. *Ficus: A Very Large Scale Reliable Distributed File System*. Ph.D. dissertation, University of California, Los Angeles, June 1991. Also available as UCLA technical report CSD-910018.
- [4] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek, and Dieter Rothmeier. Implementation of the Ficus replicated file system. In *USENIX Conference Proceedings*, pages 63–71. USENIX, June 1990.
- [5] L. B. Huston and Peter Honeyman. Disconnected operation for AFS. In *Proceedings of the USENIX Symposium on Mobile and Location-Independent Computing*, pages 1–10. USENIX, 1993.
- [6] James Jay Kistler. *Disconnected Operation in a Distributed File System*. Ph.D. dissertation, Carnegie-Mellon University, May 1993.
- [7] Puneet Kumar and Mahadev Satyanarayanan. Supporting application-specific resolution in an optimistically replicated file system. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 66–70, Napa, California, October 1993. IEEE.
- [8] Locus Computing Corporation, Inglewood, California. *PC/Interface Reference Manual*, February 1993.
- [9] Shikharesh Majumdar and Richard B. Bunt. Measurement and analysis of locality phases in file referencing behavior. In *Proceedings of Performance 86 and ACM Sigmetrics 86, Joint Conference on Computer Performance Modelling, Measurement and Evaluation*, pages 180–192, Raleigh, NC, May 1986. ACM.
- [10] Udi Manber and Sun Wu. GLIMPSE: A tool to search through entire file systems. In *USENIX Conference Proceedings*, pages 23–32, San Francisco, CA, January 1994. USENIX.
- [11] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. Technical Report UCB/CSD 85/230, UCB, 1985.
- [12] Gerald J. Popek and Bruce J. Walker. *The Locus Distributed System Architecture*. The MIT Press, 1985.
- [13] Peter Reiher, John S. Heidemann, David Ratner, Gregory Skinner, and Gerald J. Popek. Resolving file conflicts in the Ficus file system. In *USENIX Conference Proceedings*. USENIX, June 1994. To be published.
- [14] Mahadev Satyanarayanan, James J. Kistler, Lily B. Mummert, Maria R. Ebling, Puneet Kumar, and Qi Lu. Experience with disconnected operation in a mobile computing environment. In *Proceedings of the USENIX Symposium on Mobile and Location-Independent Computing*, pages 11–28, Cambridge, MA, August 1993. USENIX.
- [15] Carl D. Tait and Dan Duchamp. Detection and exploitation of file working sets. In *Proceedings of the Eleventh International Conference on Distributed Computing Systems*, pages 2–9, 1991.

Authors

Geoffrey H. Kuennen is a Ph.D. candidate in computer science at UCLA. He received the B.S. and M.S. degrees in computer science from Michigan State University in 1973 and 1974, respectively. His research interests include operating systems, distributed environments, and mobile computing. He is a member of ACM, the IEEE Computer Society, and CPSR. His Internet address is geoff@ficus.cs.ucla.edu.

Peter Reiher received his B.S. in electrical engineering from the University of Notre Dame in 1979. He received his M.S. in computer science from UCLA in 1984, and his Ph.D. in computer science in 1987. He has worked on several distributed operating systems projects. His research interests include distributed operating systems, optimistic computation, and security for distributed systems. His Internet address is reiher@ficus.cs.ucla.edu.

Gerald J. Popek has been a Professor of Computer Science at UCLA since 1973. His academic background includes a doctorate in computer science from Harvard University. He co-authored “The LOCUS Distributed System Architecture,” MIT Press, 1985, and has written more than 70 professional articles concerned with computer security, system software, and computer architectures. Dr. Popek is a principal founder of Locus Computing Corporation, the largest independent developer of UNIX-based connectivity and distributed processing software technology.

Secure Short-Cut Routing for Mobile IP

*Trevor Blackwell, Kee Chan, Koling Chang, Thomas Charuhas,
James Gwertzman, Brad Karp, H. T. Kung, W. David Li, Dong Lin, Robert Morris,
Robert Polansky, Diane Tang, Cliff Young, John Zao*

Division of Applied Sciences, Harvard University, Cambridge, MA 02138

Abstract

This paper describes the architecture and implementation of a mobile IP system. It allows mobile hosts to roam between cells implemented with 2-Mbps radio base stations, while maintaining Internet connectivity. The system is being developed as part of a course on wireless networks at Harvard and has been operational since March 1994.

The architecture scales well, both geographically and in the number of mobile hosts supported. It supports secure short-cut routing to mobile hosts using the existing Internet routing system without change. The implementation demonstrates a robust, low complexity realization of the architecture, and provides trade-off opportunities between efficiency and cost.

Measured performance of the mobile system is generally excellent. The system can handle a high rate of location updates, and routes packets almost as efficiently for mobile hosts as the Internet does for stationary hosts. We observe reasonable TCP behavior during hand-offs.

1. Introduction

Portable computers, while quite sophisticated in many ways, are hampered by the lack of support for mobility in current network protocols. The most immediate problem, the physical-layer link between computer and network, can be solved with radio. We can build on a long history of work in this area, such as Aloha [Ab 70], and more recently commercial radio hardware such as Altair [BuOdTaWh 91] and WaveLAN [Tu 88]. These radio systems provide limited geographical coverage. The cellular telephone system [Ma 79] solves this problem by tiling the world with radio base stations connected by a wired network. Our overall goal is to adapt this idea to computer networks.

The system we describe is the product of a graduate and undergraduate course on wireless networks taught at Harvard University in the 1993-94 academic year. The design was completed in the fall

of 1993. The system has been operational since March of 1994. Our experimental environment includes IBM-compatible PCs running UNIX and 2-Mbps WaveLAN spread-spectrum radio interfaces.

The next section describes goals of our system. Section 3 compares our system with other similar work. Section 4 presents the basic architecture of our system, Section 5 explains enhancement for short-cut routing, and Section 6 gives more detail about the architecture. Section 7 analyzes the security and scalability of the system. Section 8 discusses our experimental implementation and Section 9 summarizes measured performance of the system. Section 10 suggests areas for future work. The final section gives some concluding remarks.

2. System Goals

Our primary goal is that our system be transparent to users as they roam from cell to cell. A move to another office, building, or city should not affect how a user can use network services. The user should not be required to take any special action because of such a move. All the user's existing network connections should be preserved, and there should be no difference in the way new connections are created.

Performance should approach that delivered by non-mobile protocols over the same hardware. In particular, short-cut routing should be supported. A mobile IP system should not compromise the security of communication between existing wired hosts at all, and should provide the maximum practical security for mobile hosts. Packet redirection mechanisms provided for the mobile system should not be manipulable by users to deliberately cause misdelivery of packets.

We also aim at some practical goals less visible to the user. Our system should not limit the number of active mobile hosts. No administrative domain should need to know about mobile hosts from other domains, and mobile hosts should be able to roam to other domains just as they roam within a domain. We do not require changes in IP routers or non-mobile

hosts, although changes to the latter are supported to increase efficiency.

Some economic and social concerns are outside the scope of our work. We assume that different organizations are willing to provide base station service to each others' mobile hosts.

3. Background and Previous Work

A number of mobile IP systems have been implemented or proposed. All share a notion of mobile hosts (MHs), each of which keeps a constant IP address regardless of location (see Figure 1). All share the idea of radio-equipped base stations (called Foreign Agents, or FAs), which serve as temporary points of attachment to the Internet for roaming MHs. All use existing Internet routing protocols to direct packets addressed to an MH to a stationary computer (a Home Agent, or HA) capable of forwarding them to the FA to which the MH is currently attached. The fundamental differences among these systems lie in these areas:

- (1) How does an HA know where an MH is?
- (2) How can ordinary hosts send directly to an MH's current FA, avoiding the wasteful trip through the HA?
- (3) How do the mechanisms in (1) and (2) react to MH movement?

Security, scalability, and compatibility drive the choices in these three areas. A mobile IP system should not be easily tricked into redirecting packets to malicious eavesdroppers. The MH location database should not become a bottleneck as the number of MHs grows, and thus must be distributed, perhaps at the cost of some complexity to ensure consistency. Finally, mobile hosts should be able to talk to hosts that know nothing about mobility. We call hosts that send packets to an MH Correspondent Hosts (CHs). They may be ordinary and send packets to an MHs on a dog-leg route through its HA, or enhanced to use short-cut routes direct to an MH's FA.

Below we compare some other mobile IP systems to our work. We have adopted the terminology of the IETF Mobile IP Working Group [MoIP 93], though these names (MH, FA, HA, and CH) are not universally used, nor do they correspond exactly to entities in all the systems we mention. A comprehensive comparison of several of the systems is available elsewhere [MySk 93].

3.1. Columbia's System

The central theme of the Columbia's system is the notion of a single virtual subnet to which all MHs belong [IoDuMaDe 92] [IoMa 93]. Each MH uses a radio to talk to the nearest Mobile Support Router (MSR), each of which has both a radio and a wired Internet connection. Each MSR tells the IP routing system that it has an interface onto the virtual subnet, so that normal IP routers will send packets for an MH to the nearest MSR.

The system operates as follows. An MH registers with whatever MSR happens to be in radio range, and periodically reconfirms this registration. This particular MSR thus knows where the MH is. When a CH first sends a packet to the MH, the packet is forwarded to the nearest MSR by normal IP routing. If the MH is registered with that MSR, the MSR can deliver the packet to the MH directly. If not, then the MSR must find the MH. It sends a query to all the other MSRs requesting the location of the MH, and forwards the packet to whichever MSR responds. It caches the MH's location to avoid further broadcast queries.

When the MH moves to a new MSR, it informs the previous MSR of its new location. The previous MSR will cache this information and forward any packets for the MH to its new location. If the previous MSR receives a packet forwarded by another MSR, it sends that MSR a redirect specifying the MH's new location. This redirect updates that MSR's cached location for the MH.

The Columbia system's strong points are that it sends packets by efficient routes, even from computers that are not aware of mobile hosts, and that it has no unnecessary points of failure. It does not scale well, because MSRs broadcast to each other. It does have a mode of operation with improved scaling, at the cost of inefficient routing. It has no authentication, and would be vulnerable to malicious location messages.

3.2. Sony's System

Sony's system [TeUe 93] [TeTo 93] allows both CHs and intermediate routers to cache MH locations. Every MH has a permanent Virtual IP (VIP) and a Temporary IP (TIP) address. Using the normal IP routing system, Sony's scheme arranges that a packet addressed to the VIP will end up at the MH's HA, and that a packet addressed to the TIP will end up at the MH's current location.

A mobile host is allocated a TIP each time it moves to a new location; the TIP is an address on a

radio LAN at that location. The MH keeps its HA informed of its TIP. When the HA receives packets addressed to the MH's VIP, it forwards them to the MH's TIP.

When the MH sends a packet to a CH, it includes its current TIP in a special IP option [BR 89]. An enhanced CH is able to remember this TIP, and use it instead of the VIP for further communication with the MH. Packets sent to the TIP use a direct route to the MH through the Internet, avoiding the dog-leg route through the HA. Ordinary CHs ignore the option, and continue routing through the HA. When the MH moves and acquires a new TIP, it is not clear how it should notify an enhanced CH. Such a CH might continue sending to the old TIP until the MH sends it a packet containing the new TIP.

The Sony system includes routers which cache MHs' TIPs, and redirect packets sent by ordinary CHs to avoid the dog-leg through the HA. It is not clear how these caches are updated when a MH moves, especially in a network that includes ordinary routers.

The strengths of the Sony system are that it scales well and can provide efficient routing for ordinary CHs. However, its specification seems incomplete, and it provides no authentication for location updates.

3.3. IBM's System

An MH in IBM's system [RePe 92] [BhPe 93] has a permanent IP address. Each MH has an HA, and the HA tells the IP routing system that it is the gateway for its MHs. Thus when a CH sends a packet to the MH, it ends up at the HA, which will forward it to the MH. When an MH moves to a new location, it finds a nearby FA, and sends the FA's address to the MH's HA. The HA tells the MH's previous FA to forget about the MH.

When an MH sends a packet to a CH, it includes an IP Loose Source Route option [Br 89]. This option records the address of the MH's FA. The CH caches the FA address, and sends any further packets for the MH via that FA. If the MH moves, its old FA will forward packets from the CH to the MH's HA. Any reply from the MH will carry the MH's new location, allowing the CH to update its location cache.

If all Internet hosts implemented Loose Source Route correctly, IBM's system would provide efficient routing with no changes to either CHs or routers. Sadly, a dearth of correct Loose Source Route implementations thwarts this elegant system. Few systems actually remember and use the latest source route for TCP, and possibly none do so for UDP; see [MySk

93]. Source routes are not authenticated, so if implemented correctly they could be used to redirect packets arbitrarily.

3.4. Matsushita's System

Matsushita's mobile IP system [WaMa 93] [WaYoOhTa 93] is similar to the IBM scheme except in the way it provides efficient routing from CHs. When an MH moves, it acquires a temporary IP address. The MH then tries to find an FA (called a Packet Forwarding Server or PFS), and registers the FA's address with its HA (which is called the "home" PFS). The HA also receives and forwards packets sent to the MH's home address. When the HA forwards packets to the mobile host, it notifies the sending CH of the MH's current location, so the CH can then send directly to the MH.

When an MH registers a new location with the HA, the HA sends a packet to the old FA to de-register the MH and tell the old FA the mobile host's new location. If a packet for the MH arrives at the old FA, it forwards it to the MH's new location. The old FA will also inform the sending CH of the MH's new location. After a time-out period the old FA discards the new MH location, and returns MH-bound packets to the MH's HA.

The Matsushita system appears similar to our system: both include MHs, FAs, and HAs, registration at home, and support for efficient routing. However, Matsushita's Mobile IP system design does not directly address security issues or failure modes. For example, it is not clear how to perform authentication in this system, and the authors suggest repairing HA crashes by manually querying MHs for their locations. Forwarding from old PFSs to new PFSs complicates their implementation and allows forwarding loops, which must be handled specially.

3.5. Mobile IP Working Group's Proposal

This draft proposal [MoIP 93] also differs from the IBM scheme mostly in the way it provides efficient routing from CHs. Ordinary CHs always send packets to an MH via its HA. The HA, however, notices when a CH sends a packet to an MH, and notifies the CH that the MH is mobile. An enhanced CH then asks the HA for the MH's current FA, and sends further packets directly through the FA. To authenticate the HA's reply to the CH, the CH sends a random number to the HA, and the HA must supply the same number along with the MH's location. Only a router along the path between CH and HA could know this number and use

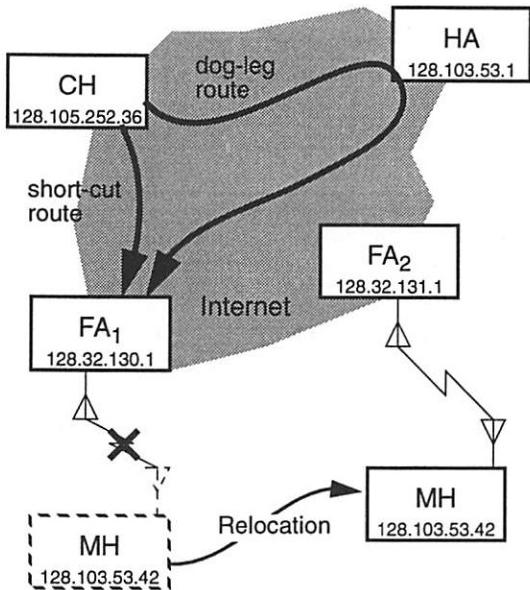


FIGURE 1. Example of mobile IP relocation, showing short-cut and dog-leg routes to the original MH location.

it to forge the HA's reply. But the CH must already trust all of those routers, since it is sending its packets through them.

The Mobile Working Group proposal is similar to our system, though they were developed independently and at roughly the same time. As a draft, it is not always complete and detailed. For instance, it is not clear how a CH determines a trustable address for an MH's HA.

A more recent draft from the Mobile Working Group [Si 94] contains more detail, but omits support for enhanced CHs. It may thus be more secure than the previous draft, and is certainly less efficient. We argue in Section 7 that enhanced CHs need not reduce security below that of the current Internet, and that therefore this omission is not necessary.

4. Basic Architecture

For explanatory purposes we consider the following scenario for our mobile IP system: a mobile host with IP address 128.103.53.42, geographically from Cambridge, Massachusetts, and under the administrative control of Harvard University, is carried to the University of California at Berkeley by its owner, Alice. Alice powers up her mobile host in Berkeley, in a wireless cell with an IP subnet number of 128.32.130. We identify the following four entities involved with providing mobile IP access to Alice's machine:

- **Mobile Host (MH):** the portable machine with wireless network hardware carried by Alice to Berkeley. It retains the IP address 128.103.53.42 regardless of its location.
- **Home Agent (HA):** the router at Harvard responsible for routing packets to mobile hosts with IP addresses in subnet 128.103.53. It remembers the locations of all MHs with addresses on that subnet. There is a single HA for each subnet which supports mobile hosts.
- **Foreign Agent (FA):** the wireless base station at Berkeley that serves as the MH's temporary attachment point to the Internet. The FA has both a radio and a wired Internet connection, and is willing to forward packets between them. An FA may serve more than one MH at the same time.
- **Correspondent Host (CH):** any host on the Internet, mobile or non-mobile, with which an MH communicates. For our example, the CH in question is in Madison, Wisconsin, with IP address 128.105.252.36.

The entities listed above are the only ones our system modifies. In particular, it uses the existing Internet routing system without any change. This is how our system behaves when the CH is not enhanced to route efficiently to mobile hosts:

- Upon arrival in Berkeley, Alice's mobile host handshakes with a nearby foreign agent. The FA arranges to route packets for the MH out its wireless interface, and the MH starts routing all its packets via the FA. The MH registers its location with its HA at Harvard, after proving its identity to the HA. The HA creates an entry in its routing table to the MH through this FA, and sets a flag indicating that packets for the MH should be encapsulated and forwarded to the FA.
- The CH in Madison sends packets to Alice's MH, at its permanent IP address. The standard Internet routing system routes these packets to the MH's HA, on the MH's home subnet. The HA looks for a route in its routing table to the MH in question, and finds the route through the FA, marked for transport by encapsulation.
- The HA encapsulates the IP packet from the Madison CH in another IP packet, and sends it to the Berkeley FA. When the FA receives this encapsulated packet, it extracts the enclosed packet, and routes it through its wireless interface to the MH.
- Alice's MH receives the packet from the FA.

- If Alice moves out of the range of the FA's radio, and into the range of another FA's radio, her MH registers the new location with its HA. The HA then starts forwarding the MH's packets via the new FA. Some packets from a CH may be forwarded by the HA to the old FA while Alice is in motion; the old FA discards them. Higher level protocols, such as TCP, should re-transmit such packets.

While the above scheme allows normal IP routing for packets from Alice's MH to Madison through the Berkeley FA and the rest of the Internet, it requires packets from the Madison CH to Alice's MH to "dog leg" through Cambridge and then double back cross-country to Berkeley. While inefficient, this routing method offers complete backward compatibility with existing Internet routers and unenhanced CH IP implementations.

The maintenance of location information for MHs by their HAs, the encapsulation of packets by a HA, and decapsulation of packets by an FA all require data structure and code modifications to the IP implementation. See Sections 6 and 8 for these and other details, such as crash recovery.

5. Enhanced Architecture for Short-cut Routing

We now present some IP enhancements made by our system that significantly improve routing efficiency from correspondent hosts to mobile hosts. Note that we maintain the invariant that existing Internet routers (those other than the foreign agent and home agent for a particular CH-MH path) require no software changes. Our goal here is to avoid the dog-leg route CH-HA-FA-MH (in our example scenario, the Madison-Cambridge-Berkeley path) in favor of the more direct CH-FA-MH (Madison to Berkeley) route for all but the first few packets from CH to MH. We modify the above behavior as follows:

- When the CH sends its first packet to the MH via the HA, the HA informs the sending CH that the MH is mobile. A non-enhanced CH ignores this notification message; such a CH continues to use dog-leg routing as outlined previously. An enhanced CH, however, asks the HA to keep it informed of the MH's location.
- The HA remembers all CHs that have subscribed to MH location updates in this way. So long as this subscription is maintained, the HA informs the CH of the MH's current FA each time the MH registers a new location.

- The CH caches the location updates from the MH's HA, installs the appropriate routes in its IP routing table (with the encapsulate flag on), and thereafter encapsulates packets bound for the MH directly to its current FA.

6. Architecture Details

We divide the architecture into four protocols: hand-off, registration, location update, and routing and encapsulation. Each of these protocols involves software that runs on more than one host; for instance, hand-off involves both FAs and MHs. The interfaces between the protocols modules on any one host are simple.

6.1. Hand-off

Each FA periodically broadcasts a beacon packet on all of its radio interfaces. If an MH is not attached to any FA and hears a beacon, it asks the FA that sent the beacon if it can attach. The FA accepts if it is not overloaded, and sends an acknowledgment. At that point the FA puts a host route for the MH in its IP routing table pointing out the radio interface, and the MH installs a default route pointing to the FA.

The MH monitors the beacons from its current FA; if it does not hear a beacon for a while, it scans for other FAs. The frequency with which FAs broadcast beacons governs how soon an MH notices that it is out of range of its current FA, and therefore how long its service will be interrupted before it acquires a new FA.

The MH periodically tells its FA that it still wants service. If the FA does not hear from the MH for a while, it deletes the route to the MH. If the FA receives an encapsulated packet for an MH for which it has no route, it silently discards the packet.

The FA provides service for an MH without any sort of authentication. This allows an unauthorized MH to send packets into the Internet via the FA, but it does not allow the MH to receive packets unless they are specifically encapsulated and sent via the FA. The only way to arrange for the MH to receive packets addressed to its IP address is by authenticated HA registration.

6.2. Registration

After the MH establishes a connection with an FA, it sends its HA a registration request. This request contains the MH's IP address and its FA's IP address. The HA replies with a randomly chosen challenge number. The MH signs the challenge along with the FA address using MD5 [Ri 92] and a secret key shared

with the HA, and sends this signature back to the HA. Upon validating the signature, the HA updates its routing table for this MH and sends back an acknowledgement. If the authentication fails, the HA replies with a denial packet. The MH must periodically re-register with its HA, in case the HA reboots and thus forgets the locations of its MHs.

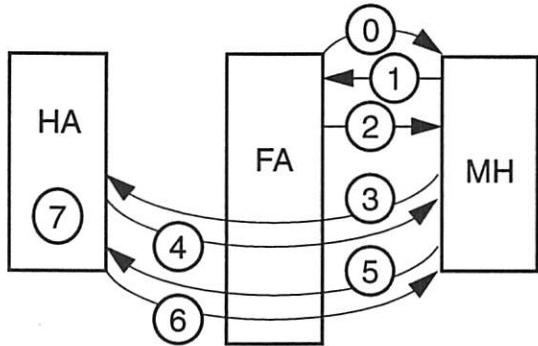


FIGURE 2. MH Hand-off and Registration. The FA periodically broadcasts beacons (0). The MH replies with an attachment request (1); the FA responds with an attachment grant (2). After attaching, the MH sends a registration request (3) to its HA. The HA replies with a challenge packet (4) to the MH. The MH sends a signed reply (5). If the reply is good, the HA sends a registration confirmation packet (6). An HA function call tells the update layer of the new MH location (7); this triggers location updates to subscribed CHs.

The HA chooses a new challenge number for an MH each time the MH registers successfully. The challenge prevents replay attacks. It also functions like a sequence number, to help the MH and HA ignore all the but the latest messages. It is especially useful when the MH changes location frequently. The MH could save one packet exchange with the HA by sending a non-repeating sequence number, rather than waiting for a challenge; we decided it would be too hard to keep the sequence numbers on the MH and HA consistent.

The HA requires stable storage to hold one registration key for each MH it serves. The key management between the HA and his MHs is straightforward as they are assumed to be under the same administrative authority.

6.3. CH Update

As described above in Section 5, the HA directly informs any CHs using dog-leg routing that the destination MH is mobile. The HA can detect when a CH talks to an MH because the Internet routes the CH's

packets via the HA. An HA limits the rate at which it notifies any one CH that an MH is mobile, since unenhanced CHs will never stop sending via the HA.

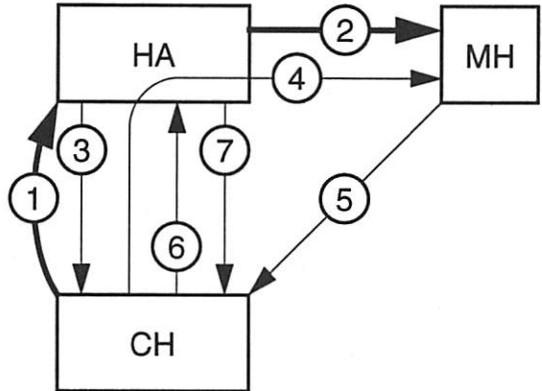


FIGURE 3. CH acquisition of direct route (FA acts only as a bridge to the MH, so it is omitted). (1) normally routed packet intercepted and (2) forwarded to MH triggers a notification message (3) to the CH. The CH asks the mobile host to name its home agent (4); after receiving the reply (5), it sends a subscription request (6) to the HA. The HA replies with a location update (7), which is then installed in the CH routing table.

In a perfect world, the HA could use one message both to inform the CH that a host is mobile and to carry the forwarding address information. Unfortunately, the CH cannot trust the contents of the notification message without creating a redirection security loophole. First, it must determine the correct address of the HA of the MH mentioned in the notification. It does this by sending a query to the MH containing a random number; the MH replies with the random number and its HA's address. The random number assures the CH that the response must have come either from the MH or from some router along the path between CH and MH. The CH must trust all routers along this path, since it sends its data through them.

After the CH has discovered the MH's HA, it sends a subscription request to the HA. The HA replies with the address of the MH's current FA. The subscription request and reply are also protected by a random number.

The relationship between HA and CH takes place under a *subscription* model. The HA remembers all of the CHs that have recently placed subscription requests. If the MH changes location, it notifies all subscribers of the new location. If a *subscription lapse time* (SLT) passes without receiving a subscription request from a particular CH, then the HA assumes that the CH no longer wishes to receive loca-

tion updates. The CH must periodically resubscribe to the HA's location update service in order to continue to receive updates. CHs determine whether a "conversation" with a particular MH is still active by checking the packet counter in the kernel routing table.

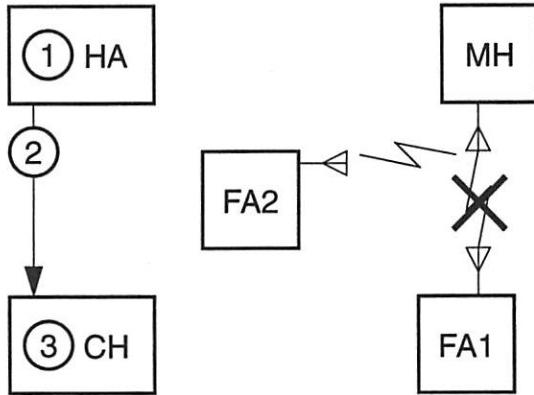


FIGURE 4. Location updates when the MH moves.
(1) Function call from registration layer triggers
(2) a new location message from the HA to all
subscribed CHs. On receipt, a CH installs the new
MH location into its route table (3).

If the CH reboots, it will begin using dog-leg routes again. The HA still sends notification messages, even for supposedly subscribed CHs, so the CH will go through the normal location update process.

If the HA reboots, it will forget all current subscribers. CHs periodically re-subscribe to help recover from such reboots. A CH removes the route to an MH if its HA fails to reply after a time-out period. This prevents permanent misdirection by a fake HA which can respond to only a finite number of the CH's subscription requests.

The MH could implement the update protocol instead of the HA. We did not choose this approach because we expect some MH operating systems will not support monitoring of packets from CHs.

6.4. Routing and Encapsulation

Both CHs and HAs need to send packets to MHs by way of FAs. They cannot directly use the regular IP routing system, since it would send packets with an MH's address to its HA. We use a simple encapsulation scheme for this, in which an IP packet for an MH is placed inside another packet, with a special IP protocol number, addressed to an FA. A flag in each routing table entry controls encapsulation. A CH encapsulates only packets that it originates, and that it knows it is sending to an MH. An HA acts as an IP

router, receiving packets from CHs that don't know an MH is mobile, encapsulating them, and forwarding them to the MH's FA.

An FA knows when it has received an encapsulated packet by looking at the IP protocol number. It strips off the outer header, and processes the packet inside almost as if it had been received in the normal way. The difference is that the FA discards the encapsulated packet if it is not addressed to an MH currently attached to the FA. This prevents routing loops.

When an MH moves, its old FA could forward packets to its new FA, rather than dropping them. This might eliminate a few lost packets. However, it is unlikely that this would eliminate all loss; for TCP, at least, a few dropped packets are very little better than many consecutive drops. In addition, it would be difficult for the old FA to authenticate the location updates that the MH would have to send it.

7. Analysis

7.1. Security

Although mobile hosts introduce some new security concerns, the fact that radio communication is easy to intercept, disrupt, and forge is not one of them. Much of the current wired Internet uses media with the same problems. Since these issues are not special to mobility, we do not attempt to address them. Systems such as Kerberos [StNeSc 88] and Privacy-Enhanced Mail [Li 89] can solve some of these problems by providing privacy and authentication between applications at either end of the network. Our aim is to maintain the Internet's current level of security for existing applications, and to help prevent denial-of-service attacks on all applications, even those with end-to-end security.

One attack we face involves a fake MH trying to register under another MH's address; the other is a fake HA sending location update messages to a CH by spoofing messages from a real HA. The second problem is particularly serious since CHs do not know which hosts are mobile; thus the same attack could be used to divert traffic from a wired host.

Security on the wired Internet is a function of what hosts are along the path between sender and receiver. More formally, if hosts A and B are communicating then a BE (Bad Element, a network-connected computer under the control of a malicious user) along the path between A and B can read all packets from A to B, forge packets from A to B, and cause packets not to be delivered to B. A host not

along the path, however, cannot read packets from A to B.

Although any host can forge a source address in the IP header, that is usually not sufficient to carry on an entire fraudulent conversation; the forger must usually see the replies to his messages to do any real harm. For instance, to send packets as part of a TCP session, the sender must use the right sender sequence number or the receiver will ignore the packets. These sequence numbers are allocated at connection setup time, using a random number generator, so that it is difficult for an attacker to guess a valid sequence number¹. Much of the security of our system is based on the assumption that attackers cannot see packets between the HA and CH for an indefinitely long period—such attackers would be able to intercept traffic directly without bothering to attack our system.

7.1.1 Security of MH-HA Registration

Our authentication scheme is implemented by two protocols. The MH-HA registration protocol authenticates the MH's identity and location to the HA. The HA-CH update protocol allows a CH to verify that it is receiving location updates from an MH's HA.

The registration message from the MH to the HA, containing the IP address of the MH's current FA, must be signed by the MH to prevent impersonation of the MH by BEs. To accomplish this, we use an MD5 signature to guarantee the authenticity of registration messages. It is unlikely that a BE could create a false registration message that the HA would accept. Replay attacks are prevented by the use of a randomly chosen challenge, which is different for each registration. Although the true MH can effectively be denied service by interception of the registration message, this is an unavoidable characteristic of any Internet connection.

We chose MD5 over some other available signature algorithms because it does not cause any interoperability problem with foreign hosts due to export restrictions. The MH and its HA must share a key which is added to the message when computing the MD5 hash, but not actually sent over the network. We generate the key and store it on both machines when the MH is first configured.

1. In fact, many Berkeley-derived TCP implementations use an easy-to-predict sequence number generator, but this should be considered broken.

7.1.2 Security of CH Location Update

Packet redirection in order to avoid dog-leg routing creates a potential security hole. A Bad Element who can forge location update messages from the HA to the CH can cause all traffic destined for an MH to be redirected to it or any other destination.

We use *tickets* to enforce the property that although BEs anywhere may be able to forge *packets* from the HA to the CH, a host can only send *valid* location update messages to the CH if it can see packets from the CH to the MH's subnet. This general security strategy is prevalent in the Internet; it is how NFS, TCP, X11's magic cookie system, and DNS achieve their security [Su 88][Ny 92][Mo 87]. No administration is necessary for this security system, and processing overhead is very small.

The CH sends the HA a subscription request asking to receive location updates for an MH, containing a ticket consisting of some randomly generated bytes X_a . Hosts not along the path between the CH and HA will not see X_a . When sending location updates, the HA includes the most recent X_a it received from that CH. The CH will only accept updates accompanied by the X_a it generated. X_a is chosen from a range large enough (2^{128} in our implementation) that a BE is unlikely to guess a valid X_a . Thus, BEs not on the path between the HA and CH cannot fool the CH into redirecting packets for a mobile host.

Recall from Section 6.3 that the CH *periodically* sends subscription requests to the HA and updates its route to the MH based on the HA's reply. Thus, even if a BE on the CH-HA path is able to see the most recent X_a , the BE can only fool the CH for a limited time until the CH's next subscription takes effect. To arrange for permanent misdirection of packets from the CH to a MH, the BE would have to be able to spoof packets on the CH-HA *continuously*. In this case, the BE would have been able to steal ordinary data packets from the CH to the MH in the first place, even without using our mobile IP system.

7.2. Scalability

We may divide the knowledge that entities in our system have about other entities into two categories:

- Static, administrative knowledge: MHs and their HA are presumed to be under control of the same administrative authority. It is assumed in our system that MHs know their home HA and that HAs know the MHs for which they route packets. A

HA and its MHs share keys for authentication purposes. This information is static in nature, and is maintained by system administrator action.

- Dynamic, online knowledge: The entities which exchange location information consist only of the MH, HA, and CH (while the FA passes along location information, it does not produce or consume it). This location information is dynamic in nature, and is automatically maintained by our system through registration and location update messages.

The limited number of parties who require knowledge in both of the above categories is a strong asset of our scheme; no entity in an administrative domain needs administrative knowledge about entities outside its domain, and no entity needs online information about any entity other than those with which it is currently communicating. This fact makes our system fundamentally scalable to a large number of entities.

The location update messages exchanged when an MH moves flow only between entities involved in communication with the MH. As shown in Section 9, the registration procedure of our system is reasonably fast.

Large numbers of MHs can be accommodated by increasing the number of HAs and home subnets; this policy for network expansion is identical to that in practice today for wired networks.

8. Implementation Notes

We have a working implementation of the system described in this paper.

Because our system involves changes to IP, we need an operating system for which we can obtain source code. We use Berkeley Software Design's UNIX (BSDI) for IBM-PC compatible computers, which includes source.

Another reason we use BSDI UNIX is that it supports the Berkeley Packet Filter [McJa 93], which can give a copy of every packet received by the system to a process. The HA software uses this to detect when a new CH starts sending packets to an MH.

Our MD5 implementation comes from the RSAREF library available from RSA Data Security, Inc.

We use WaveLAN radio interfaces [Tu 88]. WaveLAN uses spread spectrum modulation to avoid interference between nearby radios that do not wish to communicate. Radios that do wish to talk must be set to the same "code." If multiple nearby WaveLANs are

set to the same code, they can communicate peer-to-peer, though we do not currently use this feature. WaveLAN has a range of a few hundred feet and provides about 2 megabits of bandwidth per second. It uses the same frame and address format as Ethernet, and uses CSMA/CA for medium access control. The WaveLAN interfaces fit in an ISA slot in a PC. We do not currently have a truly portable radio interface using, e.g., PCMCIA, due to the difficulty of obtaining UNIX drivers for them.

8.1. Kernel Changes

Four UNIX kernel changes are needed to support the system. BSDI already allows two hosts with different IP network numbers to talk to each other over the same physical network; this situation arises when an MH talks to an FA. The only problem is with broadcasting beacon packets. The only universally acceptable IP broadcast address has all bits set. However, UNIX cannot determine on which network interface to send such a packet. We added a socket option to specify the routing table entry to be used when sending packets from a socket; in this case we would specify a route pointing to the desired interface.

We added encapsulation code to the kernel, controlled by a flag in each routing table entry. If the flag is set for the route a packet would use, the packet is encapsulated by adding a new IP header with a special protocol number. The encapsulating packet is addressed to the destination in the gateway field of the routing table entry, and is then routed in the usual way. A host knows it has received an encapsulated packet by the IP protocol number; the host strips off the encapsulating header, and processes the inner packet as if it had been received in the usual way.

This encapsulation mechanism suffices in a CH, and in an HA after an MH has registered. However, before an MH has been authenticated, its HA still needs to send it encapsulated packets. It cannot create a routing table entry for a potentially fake MH because that would divert packets away from the real MH. So we use the per-socket routing option described above during registration.

To prevent packets to un-registered MHs from being forwarded by the HA using the usual IP routing system, we added a special network interface that discards packets. The HA configures that interface with the network number used by the MHs it manages. The host routes installed for each registered MH override use of this interface. We do this in preference to giving the HA a real radio interface for its MHs so that packets for un-registered MHs are not broadcast to

anyone listening to the radio. The presence of this special interface also causes the UNIX routing daemons to announce the MHs' network number to the Internet routing system.

UNIX caches a route for every socket, which it keeps using until the route is deleted from the routing table. When a CH first receives an MH's location from an HA and installs a route for the MH, the routes cached by any sockets already connected to the MH are not affected. So while new sockets connected to the MH will use the efficient route, the first socket to send to an MH will continue to send via the HA. We have partially fixed this problem, but the UNIX IP code does not make a clean and complete solution easy.

8.2. Software Structure

Most of the software in our system runs as daemon processes, with a different type of daemon for each of the four entities (MH, FA, HA and CH). The daemons communicate across the network with UDP.

We designed and partitioned the system to make it easy for a group of students to implement as independent modules. This has worked well in most cases. For instance, we require one process to run on the MH, which combines modules for hand-off and HA registration. The interaction between them is limited to a function call made by the hand-off module to tell the registration module the IP address of the MH's current FA. The modules at both ends of each protocol, such as MH/HA registration, were implemented by the same group.

In some cases this modularity works badly. One might want to make a single computer an HA, a CH, and an FA. The three modules cannot just be executed on the same computer. All three modify the routing table, and the modifications may conflict. Worse, the FA adds routes for MHs without any authentication. Usually this is not harmful, since an MH still has to register with its HA to receive any packets. But if the FA is also the MH's HA, the MH will receive packets without registration because of the route added by the FA. We could solve this by tighter integration of the FA and HA modules.

A class of a dozen students implemented this system in a month of programming.

9. Measured Performance

We have measured performance of our mobile IP system in three areas: TCP throughput, TCP delay during hand-off, and registration speed. The computers

involved in our experiments were 66 MHz 80486 PCs. The HA, FA, and CH were connected by a single isolated Ethernet segment, with no other traffic.

Table 1 shows TCP throughput over three routes. The short-cut route performs significantly better than the dog-leg route, and approaches the performance observed on the radio link alone.

	TCP Throughput
Dog-leg Route CH->HA->FA->MH	1.1 Mbps
Short-cut Route CH->FA->MH	1.3 Mbps
Route over Radio Link Only MH -> FA or FA -> MH	1.3 Mbps

TABLE 1. TCP throughput comparisons.

Table 2 depicts the impact of hand-off time on TCP delay. Even if an MH moves between FAs with overlapping radio ranges, there will be some amount of time during which packets sent by a CH to the MH will not be delivered. This includes time for the MH to realize it has lost contact with the old FA, for the MH to scan for a new FA and attach to it, for the MH to register with the HA, and for the HA to send a location update to the CH. Some packets will be lost during this time, and must be retransmitted after an additional time-out interval by higher protocol layers such as TCP. Previous work [CaIf 93] has suggested that short hand-off times can result in disproportionately long interruptions in TCP traffic. Our experiments, summarized in Table 2, indicate that the expected interruption in service on a TCP connection is little more than twice the dead time. This is consistent with the fact that TCP doubles its retransmission time-out on each consecutive failed retransmission. Some of TCP's behavior shown in the table is due to its minimum retransmission time-out of one second and timer granularity of half a second.

Hand-off Time (Seconds)	TCP Delay (Seconds)
0.3	1.2
1.0	1.2
1.6	4.4
2.8	4.4
3.4	4.3

TABLE 2. TCP delay as a function of hand-off time.

Table 3 depicts the results of some stress tests measuring the speed of the registration process. A registration takes no more than 20 milliseconds elapsed time. This is equally divided between CPU time and transmission time.

	# Registrations per Second
MH registration to HA without CH subscribing	56
MH registration to HA with one CH subscribing	54

TABLE 3. Registration speed.

10. Future Work

Areas that warrant further investigation include improving the security of location update messages, optimizing hand-off for special cases, and load balancing for FAs in overlapping cells. We briefly explain two of these areas.

As explained earlier, our location update messages from the HA to the CH are vulnerable to spoofing and replay by persistent malicious hosts along the path between the HA and the CH. We could use digital signatures to provide better security for these updates. This would require a key and certificate management, storage, and distribution architecture to guarantee that CHs verify signatures with the correct keys.

Hand-off in our system is not particularly fast, as it requires the mobile hosts to scan channels listening for beacons from foreign agents. More coordination among FAs and MHs might allow them to locate each other faster.

11. Conclusions

The existing IP routing system makes no provision for mobile hosts; it cannot react to rapid changes in network topology, and its global knowledge of topology cannot scale to the size required to track individual hosts. We have presented the architecture and implementation of a solution to this problem. It makes use of IP routing and the Internet infrastructure without modification. It maintains a database of mobile host locations, partitioned in a way that allows scaling. It is backward-compatible with existing hosts, but gives the option of increasing routing efficiency by adding short-cut routing to host IP software. Neither the location database nor the host IP modifications decrease security below the level provided by today's Internet.

Our system turns out to be quite close to the overall direction outlined in a recent draft [MoIP 93] of the IETF Mobile Working Group. It appears that we have one of the first working implementations of the architectural approach being pursued by the Group. Our implementation demonstrates the practicality of the approach, including secure short-cut routing.

Acknowledgments

Digital Equipment Corporation lent us WaveLAN interfaces and provided equipment discounts for the project's PCs. Anders Klemets wrote the original version of the WaveLAN driver while at CMU. NCR has allowed others to use this driver. Motorola gave us Altair radio network units. Bell-Northern Research supported our efforts via unrestricted research grants. Hewlett-Packard and IBM gave us laptops. We thank them for making our work possible.

References

- [Ab 70] N. Abramson, "The ALOHA System - Another Alternative for Computer Communications." *Proceedings, Fall Joint Computer Conference*, 1970.
- [BhPe 93] Pravin Bhagwat & Charles Perkins, "A Mobile Network System based on Internet Protocol (IP)", USENIX Symposium on Mobile and Location Independent Computing, Aug, 1993, Cambridge, MA, 69-82.
- [Br 89] R. Braden, "Requirements for Internet Hosts - Communication Layers", RFC 1122, Oct 1989.
- [BuOdTaWh 91] Dale Buchholz, Paul Odlyzko, Mark Taylor, and Richard White, "Wireless In-Building Network Architecture and Protocols," IEEE Network Magazine, Nov. 1991.
- [CaIf 93] Ramon Caceres & Liviu Iftode, "Effects of Mobility on Reliable Transport Protocols," Matsushita Information Technology Labs, TR73-93.
- [IoDuMaDe 92] John Ioannidis, Daniel Duchamp, Gerald Maguire, & Steve Deering, "Protocols for Supporting Mobile IP Hosts", Internet Draft, June 1992.
- [IoMa 93] John Ioannidis & Gerald Maguire, "The Design and Implementation of a Mobile Internetworking Architecture", Proc. of Winter USENIX, Jan 1992, San Diego, CA, p. 491-502.

- [Li 89] J. Linn, "Privacy Enhancement for Internet Electronic Mail, Parts I, II, and III," RFC 1113-1115, January 1989.
- [Ma 79] V. H. MacDonald, "The Cellular Concept," Bell System Technical Journal, vol. 58 no. 1, part 3, Jan. 1979.
- [McJa 93] Steven McCanne and Van Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," Winter 1993 USENIX Conference.
- [Mo 87] P. V. Mockapetris, "Domain Names: Implementation and Specification," RFC 1035, 1987.
- [MoIP 93] Mobile IP Working Group, "Routing Support for IP Mobile Hosts", Internet Draft, Dec 1993.
- [MySk 93] Andrew Myles & David Skellern, "Comparison of Mobile Host Protocols for IP", available from authors. (andrewm@mpce.mq.edu.au)
- [Ny 92] A. Nye, ed., "The X Window System, Volume 0: X Protocol Reference Manual, Third Edition," O'Reilly and Associates, Sebastopol, California, 1992.
- [RePe 92] Yakov Rekhter & Charles Perkins, "Optimal Routing for Mobile Hosts using IP's Loose Source Route Option", Internet Draft, Oct 1992.
- [Ri 92] R. L. Rivest, "The MD5 Message-Digest Algorithm," RFC 1321, April 1992.
- [Si 94] W. A. Simpson (ed.), "IP Mobility Support," IETF Internet Draft, March 1994.
- [StNeSc 88] J. Steiner, C. Neuman, and J. I. Schiller, "Kerberos: An Authentication Service for Open Network Systems," in *Proc. Winter USENIX Conference*, Dallas, 1988.
- [Su 88] Sun Microsystems, "NFS: Network File System Protocol Specification," RFC 1094, March, 1988.
- [TeTo 93] Fumio Teraoka & Mario Tokoro, "Host Migration Transparency in IP Networks: The VIP Approach", Comp. Comm. ACM, Jan 1993, p. 45-65.
- [TeUe 93] Fumio Teraoka & Keisuke Uehara, "The Virtual Network Protocol for Host Mobility", Internet Draft, July 1993.
- [Tu 88] Bruch Tuch, "Development of WaveLAN, an ISM Band Wireless LAN," AT&T Technical Journal, July/August 1993.
- [WaYoOhTa 93] Hiromi Wada, Takashi Yozawa, Tatsuya Ohnishi, & Yasunori Tanaka, "Mobile Computing Environment Based on Internet Packet Forwarding", Proc. of Winter USENIX, Jan 1993, San Diego, CA, p. 503-517.
- [WaMa 93] Hiromi Wada & Brian Marsh, "Packet Forwarding for Mobile Hosts", Internet Draft, July 1993.

Author Information

James Gwertzman and Diane Tang are undergraduates at Harvard.

Trevor Blackwell, Kee Chan, Koling Chang, Thomas Charuhas, Brad Karp, W. David Li, Dong Lin, Robert Morris, Robert Polansky, Cliff Young and John Zao are graduate students at Harvard.

H. T. Kung is Gordon McKay Professor of Electrical Engineering and Computer Science at Harvard. He is the instructor of the course (CS96) on wireless networks that has developed the system described herein.

The postscript file of the current version of this paper is available via anonymous FTP from virtual.harvard.edu:/pub/cs96/usenix94.ps.

THE USENIX ASSOCIATION

The USENIX Association is a not-for-profit membership organization of those individuals and institutions with an interest in UNIX and UNIX-like systems and, by extension, C++, X windows, and other programming tools. It is dedicated to:

- * sharing ideas and experience relevant to UNIX or UNIX inspired and advanced computing systems,
- * fostering innovation and communicating both research and technological developments,
- * providing a neutral forum for the exercise of critical thought and airing of technical issues.

Founded in 1975, USENIX is well known for its twice-a-year technical conferences, accompanied by tutorial programs and vendor displays. Also sponsored are frequent single-topic conferences and symposia. USENIX publishes proceedings of its meetings, the bi-monthly newsletter *:login:*, the refereed technical quarterly, *Computing Systems*, and has expanded its publishing role in cooperation with the MIT Press with a book series on advanced computing systems. The Association actively participates in various ANSI, IEEE and ISO standards efforts with a paid representative attending selected meetings. News of standards efforts and reports of many meetings are reported in *:login:*.

SAGE, the System Administrators Guild

The System Administrators Guild (SAGE) is a Special Technical Group within the USENIX Association, devoted to the furtherance of the profession of system administration. SAGE brings together system administrators for professional development, for the sharing of problems and solutions, and to provide a common voice to users, management, and vendors on topics of system administration.

A number of working groups within SAGE are focusing on special topics such as conferences, local organizations, professional and technical standards, policies, system and network security, publications, and education. USENIX and SAGE will work jointly to publish technical information and sponsor conferences, tutorials, and local groups in the systems administration field.

To become a SAGE member you must be a member of USENIX as well. There are six classes of membership in the USENIX Association, differentiated primarily by the fees paid and services provided.

USENIX Association membership services include:

- * Subscription to *:login:*, a bi-monthly newsletter;
- * Subscription to *Computing Systems*, a refereed technical quarterly;
- * Discounts on various UNIX and technical publications available for purchase;
- * Discounts on registration fees to twice-a-year technical conferences and tutorial programs and to the periodic single-topic symposia;
- * The right to vote on matters affecting the Association, its bylaws, election of its directors and officers;
- * The right to join Special Technical Groups such as SAGE.

Supporting Members of the USENIX Association:

ANDATACO
ASANTÉ Technologies, Inc.
Frame Technology Corporation
Matsushita Electrical Industrial Co., Ltd.
Network Computing Devices, Inc.

OTA Limited Partnership
Quality Micro Systems, Inc.
Tandem Computers, Inc.
UNIX System Laboratories, Inc.
UUNET Technologies, Inc.

SAGE Supporting Member:
Enterprise Systems Management Corporation

For further information about membership, conferences or publications, contact:

The USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Email: office@usenix.org
Phone: +1-510-528-8649
Fax: +1-510-548-5738

ISBN 1-880446-62-6